AFRL-RY-WP-TR-2016-0046

MAINTAINING ENTERPRISE RESILIENCY VIA KALEIDOSCOPIC ADAPTION AND TRANSFORMATION OF SOFTWARE SERVICES (MEERKATS)

Roxana Geambasu, Dimitris Mitropoulos, Simha Sethumadhavan, and Junfeng Yang
Columbia University

Angelos Stravrou and Dan Fleck
George Mason University

Matthew Elder and Azzedine Benameur
Symantec

APRIL 2016
Final Report

AIR FORCE RESEARCH LABORATORY
SENSORS DIRECTORATE
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7320
AIR FORCE MATERIEL COMMAND
UNITED STATES AIR FORCE

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nationals.

AFRL-RY-WP-TR-2016-0046 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

// Signature//

TOD J. REINHART
Program Manager
Avionics Vulnerability Mitigation Branch
Spectrum Warfare Division

// Signature//

DAVID HAGSTROM, Chief
Avionics Vulnerability Mitigation Branch
Spectrum Warfare Division

// Signature//

NEERAJ PUJARA, Chief (Acting)
Spectrum Warfare Division
Sensors Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

*Disseminated copies will show "//Signature//" stamped or typed above the signature blocks.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
April 2016	Final	30 September 2011 – 31 December 2015

4. TITLE AND SUBTITLE
MAINTAINING ENTERPRISE RESILIENCY VIA KALEIDOSCOPIC ADAPTION AND TRANSFORMATION OF SOFTWARE SERVICES (MEERKATS)

5a. CONTRACT NUMBER
FA8650-11-C-7190

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
62303E

6. AUTHOR(S)
Roxana Geambasu, Dimitris Mitropoulos, Simha Sethumadhavan, and Junfeng Yang (Columbia University)
Angelos Stravrou and Dan Fleck (George Mason University)
Matthew Elder and Azzedine Benameur (Symantec)

5d. PROJECT NUMBER
3000

5e. TASK NUMBER
YW

5f. WORK UNIT NUMBER
Y03X

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Columbia University
Sponsored Projects Administration
615 West 131st Street, Rm 254, Mail Code
New York, NY 10027

George Mason University
Symantec

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory
Sensors Directorate
Wright-Patterson Air Force Base, OH 45433-7320
Air Force Materiel Command
United States Air Force

DARPA/I2O
675 N Randolph Street
Arlington, VA 22203

10. SPONSORING/MONITORING AGENCY ACRONYM(S)
AFRL/RYWA

11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S)
AFRL-RY-WP-TR-2016-0046

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

13. SUPPLEMENTARY NOTES
The U.S. Government is joint author of the work and has the right to use, modify, reproduce, release, perform, display or disclose the work. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. Report contains color.

14. ABSTRACT
We have designed, implemented, evaluated, and in some cases deployed a set of new technologies that add continuous change, deception, and unpredictability to cloud environments as a way to increase their resilience to a broad spectrum of threats. Our work makes significant advances along five major directions: (1) continuous migration technologies that can enable for the first time the swift migration of cloud-resident services and data either in response to an attack or continuously so as to present a moving-target defense; (2) cloud information flow tracking technologies that can track cloud-resident data at larger scales than ever before; (3) misinformation and decoy technologies that can automatically generate deceptive information - bogus information that appears genuine - so as to confuse, bait, and track attackers; (4) cloud monitoring and self-healing technologies that integrate information from many sensors to detect complex, multi-stage attacks; (5) stable multithreading technologies that reduce the security risks posed by concurrent programs by ensuring that programs take only a few pre-checked, safe schedules during execution; and (6) hardware-enhanced execution memoization techniques that enable efficient execution in highly replicated environments.

15. SUBJECT TERMS
cloud resilience, moving-target defense, decoy information, anomaly detection, stable multithreading, memoization

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT:	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON (Monitor)
a. REPORT	b. ABSTRACT	c. THIS PAGE			Tod Reinhart
Unclassified	Unclassified	Unclassified	SAR	60	19b. TELEPHONE NUMBER *(Include Area Code)* N/A

CONTENTS

LIST OF FIGURES ii

LIST OF TABLES ii

1 SUMMARY 1

2 INTRODUCTION 2

3 METHODS, ASSUMPTIONS, AND PROCEDURES 4

4 RESULTS AND DISCUSSION 5
4.1 Continuous Migration Technologies 5
4.1.1 MOTAG (a.k.a. MiSS): Migration-based Denial of Service Defense 5
4.1.2 EPC: Urgent Virtual Machine Migration 12
4.1.3 CleanOS (a.k.a. Evade): Continuous Data Migration 15
4.2 Cloud Information Flow Tracking Technologies 20
4.2.1 CloudFence: Data Flow Tracking for Clouds 21
4.2.2 Cloudopsy: Visualization Tool for Cloud Data Flow 23
4.2.3 Data Provenance with Dynamic Instrumentation 24
4.3 Misinformation and Decoy Technologies 25
4.3.1 DIGIT: Computational Decoys 25
4.3.2 Novo: Document and App Decoys (Commercialized) 26
4.3.3 SAuth: Password Decoys 26
4.4 Cloud Monitoring and Self-Healing Technologies 29
4.4.1 DMCC: Distributed Monitoring and Cross Checking 29
4.4.2 CRP: Self-Healing Multitier Defense 30
4.4.3 Virtual Application Partitioning 30
4.5 Stable Multithreading Technologies 32
4.5.1 Stable Multithreading 33
4.5.2 Crane: Correctly Replicating Nondeterministic Executions 34
4.6 Hardware-Enhanced Memoization Technologies 39
4.6.1 Value Reuse Measurement 39
4.6.2 Compiler-Enhanced Memoization 41
4.6.3 Hardware-Enhanced Memoization 42
4.7 New Attacks 44
4.7.1 ARC: Protecting against HTTP Parameter Pollution Attacks 44
4.7.2 CellFlood: A New Attack Against Tor Onion Routers 45

5 CONCLUSIONS 47

6 PUBLISHED PAPERS 48

REFERENCES 50

LIST OF KEY ABBREVIATIONS AND ACRONYMS 59

LIST OF FIGURES

1 Shuffling Algorithm 8
2 Shuffling Evaluation 10
3 MOTAG Prototype 11
4 Client Migration Time 12
5 QEMU-KVM Live Migration vs. Post-Copy Migration 13
6 EPC vs. Live Migration 15
7 CleanOS Architecture 17
8 CleanOS Audit Service 19
9 Data Exposure 19
10 CloudFence Architecture 22
11 Cloudopsy Screenshot 23
12 Synergy-Enhanced Authentication 28
13 Multithreading Alternatives 33
14 Crane Performance 38

LIST OF TABLES

1 Notations 9
2 Time Bubbles in Crane 38

1 SUMMARY

This work investigates a new vision for increasing the resilience of computing clouds by elevating continuous change, evolution, and misinformation as first-rate design principles of the cloud's infrastructure. The work is motivated by the fact that today's clouds are very static, uniform, and predictable, allowing attackers who identify a vulnerability in one of the services or infrastructure components to spread their effect to other, mission-critical services. Our goal is to integrate into clouds a new level of unpredictability for both their services and data so as to both impede an adversary's ability to achieve an initial system compromise and, if a compromise occurs, to detect, disrupt, and/or otherwise impede his ability to exploit this success.

As a step toward this vision, we designed, implemented, evaluated, and in some cases deployed a broad set of new technologies that add continuous change, deception, and unpredictability to cloud environments. These technologies present significant advances along five major directions: (1) *continuous migration technologies* that can enable for the first time the swift migration of cloud-resident services and data either in response to an attack or continuously so as to present a moving-target defense; (2) *cloud information flow tracking technologies* that can track cloud-resident data at larger scales than ever before, enabling cloud users (e.g., service administrators) to audit the flow of their information in the cloud; (3) *misinformation and decoy technologies* that can automatically generate deceptive information – bogus information that appears genuine – so as to confuse, bait, and track attackers; (4) *cloud monitoring and self-healing technologies* that can integrate information from many sensors spread across the cloud to detect complex, multi-stage attacks; (5) *stable multithreading technologies* that can reduce the security risks posed by concurrent programs by ensuring that upon every execution, a program takes one of a few pre-checked schedules that have already been validated as safe; and (6) *hardware-enhanced memoization technologies* that enable efficient execution of highly replicated environments.

We have already started a transition of some of these technologies (notably the decoy technologies) as part of this project. While more work remains to be done to roll out all of these technologies into a production cloud environment; we believe that in the future, they will become the bases for crucial components of future resilient clouds.

2 INTRODUCTION

Our MEERKATS project investigates a new vision for increasing the resilience of computing clouds by elevating continuous change, evolution, and misinformation as first-rate design principles of the cloud's infrastructure. Our work is motivated by the fact that today's clouds are very static, uniform, and predictable, allowing attackers who identify a vulnerability in one of the services or infrastructure components to spread their effect to other mission-critical services. Our goal is to enable cloud systems to achieve unpredictability at multiple levels for both their services and data so as to both impede an adversary's ability to achieve an initial system compromise and, if a compromise occurs, to detect, disrupt, and/or otherwise impede his ability to exploit this success.

As a step toward this vision, we designed, implemented, evaluated, and in some cases deployed a broad set of new technologies that add continuous change, deception, and unpredictability to cloud environments. Described across more than 30 papers we published as part of MEERKATS, these technologies present significant advances in several major directions:

1. **Continuous Migration Technologies:** We developed technologies that can enable the swift migration of cloud-resident services and data either in response to an attack or continuously so as to present a moving-target defense. Our research indicates that: (1) A moving target defense can be used to contain the effect of a distributed denial of service (DDoS) attack. (2) By designing a virtual machine migration mechanism specifically for the purpose of swift migration, one can effectively implement a continuous or highly reactionary migration scheme (something that we show is impossible with today's live migration technologies). And (3), a careful design of the operating system running on each cloud server, which monitors all data resident on that machine and automatically encrypts it whenever it is not under active use, can limit the exposure of sensitive information by over 90% to an attack perpetrated against that machine. These findings, along with the systems and evaluation methodologies we used to arrive at these conclusions, are presented in Section 4.1.

2. **Cloud Information Flow Tracking Technologies:** Being able to monitor and audit where information flows in a cloud is crucial to detecting information leaks and data exfiltration attacks. As part of MEERKATS, we developed the first comprehensive framework for tracking information flows in large-scale computing clouds. Our framework consists of three integrated components: (1) A cloud-wide information flow tracking system that provides the scale necessary to track the information of many users across many services and ma- chines. (2) An information flow visualization system that enables auditing of these flows in a user-friendly manner; it addresses scaling and data visualization challenges. And (3), we developed a universal data provenance framework that, for the first time, gathers provenance information for real-world applications without any code modifications. The systems we developed, as well as how we integrated and evaluated them, are presented in Section 4.2.

3. **Misinformation and Decoy Technologies:** Adversary misinformation and deception have long been critical defense tools in military engagements, however today's clouds typically lack these components in their defense tool chains. For this reason, when an adversary successfully breaks into a system and steals some data (e.g., the password database, as it often happens), all of that

data is compromised without any possibility of remediation. As part of MEERKATS, we developed a set of misinformation and decoy technologies capable of automatically generating deceptive information – information that appears genuine but that is entirely bogus. Our technologies help confuse adversaries, make them waste effort and time on filtering bogus information, and can even help identify attackers by providing a long-term monitoring tool for leaked information. A key challenge in developing misinformation technologies is to create bogus information that looks real upon human or machine inspection. We have developed realistic decoy technologies for several specific domains: (1) *computational decoys*, which create real-looking bogus cloud service requests so as to spur fake computations in a cloud; (2) *document and application decoys*, which create real-looking, bogus documents that are placed on a user's machine to catch illicit accesses; and (3) *password decoys*, which can be added to password databases to address whole-database leaks. Parts of our decoy generation technology are being commercialized through support from DARPA by Allure, Inc. The research systems we developed are presented in Section 4.3.

4. **Cloud Monitoring and Self-Healing Technologies:** A resilient cloud must be able to monitor its components to detect emerging attacks before they inflict damage and to either defeat them through self-healing or to direct other defensive components to increase their focus on specific, vulnerable components. As part of MEERKATS, we developed a set of cloud-wide monitoring, which allow the efficient and scalable integration of information from many sensors spread across the cloud to detect anomalous behavior that can indicate attacks. A key contribution is our development of the first detection mechanism for multi-stage attacks, which correlates attacks over time. This and other monitoring systems we developed are presented in Section 4.4.

5. **Stable Multithreading Technologies:** A significant risk for the security of cloud systems stems from the pervasive deployment of multithreaded programs. These programs are notoriously difficult to write correctly and safely, as well as to test and debug. And unfortunately, bugs in multithreaded programs can lead to serious vulnerabilities such as time of check time of use attacks and memory corruptions, which attackers can exploit to gain unauthorized information, corrupt critical data structures, and execute code. Hence, a key part of our MEERKATS effort has been to develop new approaches for safer and more reliable multi-threaded programming. Our key innovation in this space is the notion of *stable multithreading*, which lets a program follow only schedules known to be safe. We have developed a number of stable multithreading systems, which both implement and leverage the concept, all of which are described in Section 4.5.

6. **Hardware-Enhanced Memoization Technologies:** A key challenge in MEERKATS is to perform all of the redundant, decoy executions necessary to confuse an adversary without consuming huge amounts of extra resources. We thus investigated the notion of program memoization and developed *CHAMP*, a hardware/software framework for efficient program memoization. A key finding is that solely software enhancements for memoization are insufficient and that small hardware can provide much better performance. Section 4.6 discusses these aspects.

7. **New Attacks:** Finally, we have investigated a number of new kinds of attacks that are possible against critical components of today's web services. We discuss these in Section 4.7.

We believe that the cloud resilience technologies we have developed over the course of this program will be crucial components of future resilient clouds.

3 METHODS, ASSUMPTIONS, AND PROCEDURES

In the course of the MEERKATS program, we have designed, built, evaluated, and at times deployed, these technologies as standalone systems. This enables their testing, deployment, and commercialization in diverse use cases where they can be beneficial. For example, decoy documents and applications are valuable both in cloud environments and in mobile-user use cases; our commercialization effort currently focuses on the latter use case. In the future, we hope to additionally implement a coherent cloud architecture that combines these technologies to increase resiliency against a wide range of attacks.

We took a systems approach to designing, prototyping, and evaluating each technology. Generally speaking, the technologies are designed for use in Linux (or variants of it), and are evaluated along multiple different dimensions, depending on the specific goal of the technology: e.g., security benefits, performance, scalability, availability, and usability for programmers and users.

Given the diversity of the technologies we developed, there is not one single set of assumptions or threat model that applies to all cases. Subsequent section, which describe each technology, provide the necessary background into the specific problem or threat each technology addresses, what assumptions it makes, how it works, and how we have evaluated it.

4 RESULTS AND DISCUSSION

4.1 Continuous Migration Technologies

As a first critical component of a resilient cloud, we developed a series of technologies to support continuous and lightweight migration of both services and data. Our purpose was to enable moving-target defenses to withstand a number of attacks, ranging from distributed denial of service (DDoS) to complete server compromises (e.g., rootkits or physical takeover). Along these lines, we developed three systems:

1. We demonstrated the effectiveness of a moving-target defense to increase resilience in the face of DDoS attacks [48]. We implemented a system, called *MOTAG* (formerly known as MiSS), to implement this moving-target defense for Amazon Elastic Compute Cloud (EC2).

2. We developed a new type of virtual machine migration, called *urgent migration*, which optimizes for swift migration of heavily loaded virtual machines (VMs) upon request [5]. The system behaves better than live migration in high-load situations and can be used to swiftly move VMs that are suspected of being under attack (e.g., DDoS) to free up the host's resources of the remaining, benign VMs.

3. We developed *CleanOS* (formerly known as Evade), a new operating system designed to continuously cleanse itself of sensitive data [88]. CleanOS tracks the use of this data and whenever the data is not under active use, it is encrypted with a key that is escrowed on a trusted, resilient key service. That service audits all accesses to the keys. CleanOS ensures that at any time, any machine in the cloud has the minimal amount of sensitive data exposed in it in anticipation of server compromise.

4.1.1 MOTAG (a.k.a. MiSS): Migration-based Denial of Service Defense

We developed a cloud-based moving target defense system to protect vital Internet services from distributed denial of service attacks (DDoS). This corresponds to the MiSS component of the MEERKATS proposal, which we have renamed MOTAG.

Our moving target defense mechanism defends both authenticated services and open web services from Internet DDoS attacks. Our mechanism employs a group of dynamic, hidden proxies to relay traffic between clients and servers. By continuously replacing attacked proxies with backup proxies and reassigning (shuffling) the attacked clients onto new proxies, innocent clients are segregated from malicious insiders through a series of shuffles. To accelerate the process of insider segregation, we designed an efficient greedy algorithm which is proven to have near optimal performance empirically [48]. We built and deployed a demonstration capability on Amazon EC2 to test our ideas in a real cloud environment.

In addition, we modeled and quantified the insider quarantine capability of the greedy algorithm to enable defenders to estimate the resources required to defend against DDoS attacks and meet defined quality of service (QoS) levels under various attack scenarios. Simulations were then performed which confirmed the theoretical results and showed that our mechanism is effective in mitigating the

effects of an insider-assisted DDoS attack. The simulations also demonstrated that the overhead introduced by the shuffling procedure is low.

We also researched the costs versus defensive capabilities to enable the government to make informed decisions regarding uptime versus cost. Through our use of cloud-based services we are able to maintain a low cost steady-state posture when the system is not under attack, and quickly ramp up the number of servers to handle attack loads while isolating attackers from benign clients.

Through this program we have concluded that a moving target-based approach is both feasible and cost-effective to mitigate DDoS attacks. The system prototype and algorithms developed can be used as a reference to build and test a full production capability. Our team plans to continue to work towards a full production capability.

Problem Statement. Distributed Denial-of-Service (DDoS) attacks are a rapidly growing problem which poses an immense threat to the Internet. Akamai has reported a significant increase in the prevalence of large-scale distributed denial-of-service (DDoS) attacks in recent years [89]. In 2010, the largest reported bandwidth achieved by a flood-based DDoS attack reached 100 Gbps. Even as the bandwidth of attacks has increased, the cost of performing a DDoS attack has turned out to be surprisingly low. A Trend Micro white paper [69] reported that the price for a 1-week DDoS attack could be as low as $150 on the Russian underground market.

A number of mechanisms have been proposed in the past to prevent or mitigate the impact of DDoS attacks. Filtering-based approaches [65, 37, 62] use ubiquitously deployed filters that block unwanted traffic sent to the protected nodes. Capability-based defense mechanisms [10, 97, 99, 63] endeavor to constrain resource usage by the senders to beneath a threshold defined by the defended system. Secure overlay solutions [52, 85, 8, 86, 66, 34] interpose a network of proxies that redirect packets between clients and the protected nodes and are designed to absorb and filter out attack traffic. All these mechanisms are effective to varying degrees, but these static defense mechanisms either rely on the global deployment of additional functionalities on Internet routers, or require large, robust, virtual networks designed to withstand the ever-larger attacks. Due to the large investment, vulnerability to sophisticated attacks such as sweeping [85] and adaptive flooding attacks [8], the development of a novel, effective, efficient, and low cost defense mechanism continues to be a high priority, but elusive goal.

MOTAG Overview. In this program, we developed *MOTAG*, a MOving Target defense mechanism AGainst Internet DDoS attack. This dynamic DDoS defense mechanism implements a moving scheme of moving proxy nodes to protect centralized online services. In particular, *MOTAG* offers DDoS resilience for clients of security sensitive services such as military command and control, online banking and e-commerce. *MOTAG* employs a layer of secret moving proxy nodes to relay communications between clients and the protected application servers.

The proxy nodes in *MOTAG* have two important characteristics. First, the proxy nodes are “secret” in that their IP addresses are concealed from the general public and are exclusively known only to legitimate clients and only after successful authentication in the authenticated version of our system.

In order to avoid unnecessary information leakage, each client is provided with the IP address of only a single proxy node at any given time. Existing proof-of-work schemes [12, 32, 93, 75] are employed to protect the client authentication channel when authentication is used. Second, the proxy nodes are "moving". As soon as an active proxy node is attacked, it is replaced by a set of alternate proxy nodes instantiated at a different IP address, and the clients associated with the attacked proxy node are migrated to alternative proxy node(s). We show that this migration to "secret" proxy nodes not only enables the *MOTAG* mechanism to mitigate brute-force DDoS attacks, but also provides a means to discover and isolate malicious insiders designed to divulge the location of the secret proxy nodes to external attackers. The malicious insiders are isolated via a shuffling process that reassigns and migrates clients through sequential sets of instantiated of proxy nodes. We developed algorithms to (1) accurately estimate the number of insiders and (2) dynamically determine the client-to-proxy assignment that will "save" the largest number of legitimate clients after each shuffle.

Unlike previously proposed DDoS defense mechanisms, *MOTAG* does not rely on global adoption on Internet routers or collaboration across different ISPs to function. Also *MOTAG* neither depends on resource-abundant overlay network to out-muscle high bandwidth attacks nor uses filters to provide fault tolerance. Instead, we take advantage of our proxies' secrecy and mobility to fend off powerful attackers. Employing the *MOTAG* DDoS defense mechanism requires lower deployment costs while offering substantial defensive agility, resulting in effective and cost-efficient DDoS protection.

Threat Model. In the Mission-oriented Resilient Cloud (MRC) program we developed moving target defenses for both open and general-purpose web services as well as security sensitive online services which require authentication. In both cases the shuffling and insider detection remain the same, the only difference is the initial method to allocate users to proxies.

We assume the availability a cloud environment with sufficient computing power and bandwidth to instantiate numerous backup proxy nodes. Since only a small group of proxies are active at any time, a cloud environment in which customers are charged only for running instances would be ideal to avoid extensive operational costs. We further assume that although powerful attackers with a high aggregate bandwidth are capable of simultaneously overwhelming many standalone machines on the Internet, attackers cannot saturate the well-provisioned Internet backbone links of ISPs, data centers, and cloud service providers.

We also assume that attackers, in case of uncertainty, can first perform a reconnaissance attack (e.g., IP and port scanning) to pinpoint targets for the subsequent flooding attack. In the authenticated system, with knowledge of the *MOTAG* mechanism, attackers could attempt to flood the authentication channel through which the legitimate clients are admitted. However, since it is significantly harder for attackers to pass strong authentication by brute force and reach the proxies as legitimate clients, some attackers will attempt to uncover the network locations of proxy nodes, and may plant "insiders" by compromising legitimate clients or eavesdropping on legitimate clients' network connections. However, the number of such insiders in a protected system is assumed to be limited.

The Shuffling Process. To understand the defense system, first you must understand how shuffling is used to isolate attackers. This section presents an in-depth analysis of the development of the

proof-of-concept client-to-server shuffling mechanism. The goal is to separate benign clients from persistent bots that follow the moving replica servers to continue a DDoS attack. This is hard to achieve using an engineering method because such persistent bots may behave exactly like benign clients. With our solution, new replica servers are dynamically instantiated during a DDoS attack to replace the ones bombarded by bots. The shuffling operation refers to a structured method of re-assigning the affected clients from the attacked replicas to the new shuffling replica servers. Replica servers that are currently not under attack do not participate in a shuffling operation; although they may start to if they are bombarded later.

To illustrate the idea of client-to-server shuffling, we show the steps for one round of shuffling through the example in Figure 1. The initial state of the system is displayed on the left, where six clients $\{C1, C2, \ldots, C6\}$ are randomly assigned to two replica servers $\{RS\ 1, RS\ 2\}$. Among these clients, $C3$ and $C4$ are persistent bots attempting to attack the protected service. Like benign clients, the persistent bots have gone through the redirection steps to reach the replica servers. After successfully locating the service, the persistent bots instruct the naive bots to bombard both $RS\ 1$ and $RS\ 2$. Our solution aims to mitigate such an attack by gradually segregating the persistent bots from the benign clients.

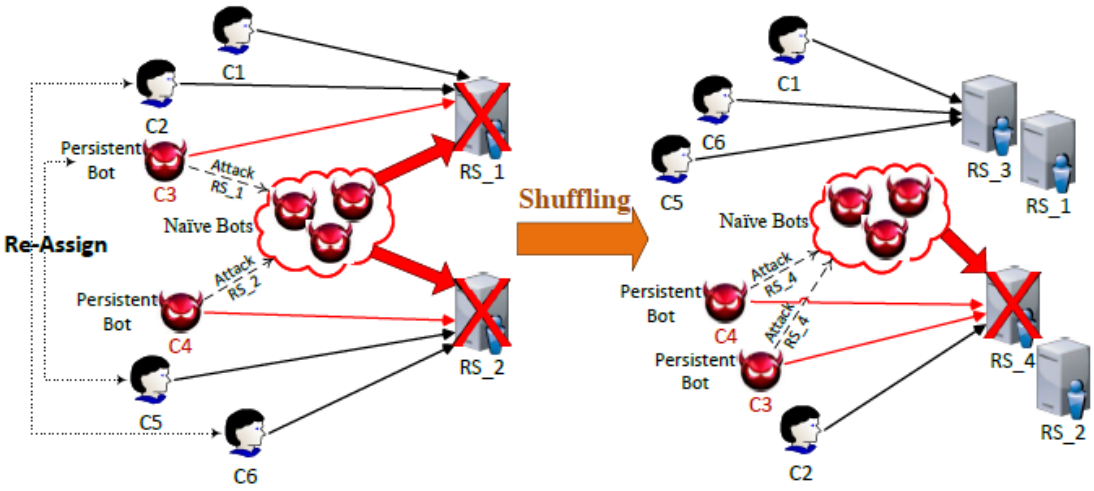


Figure 1: Shuffling Algorithm

For this purpose, we instantiate new replica servers $\{RS\ 3, RS\ 4\}$ in the cloud to replace the attacked server instances. Then, we shuffle the clients' assignments onto the new replica servers. One possible shuffle is to swap the assignments of $C2$ and $C6$, $C3$ and $C5$ from the previous allocation scheme. As is indicated by the right half of Figure 1, this client-to- server shuffling will segregate all persistent bots onto $RS\ 4$, leaving $RS\ 3$ out of the bots' vision. As a result, the benign clients assigned to $RS\ 3$ will be separated from the bots and saved from the ongoing DDoS attack. At this point, $RS\ 3$ and the saved clients will stop participating in the shuffling operation. New shuffling replica servers can be instantiated to shuffle the clients on RS_4.

To maximize the number of benign clients saved from each shuffle, the coordination server needs to make informed and optimal decisions based on the status quo. To that end, we developed algorithms

to realize fast bot/attacker segregation. The controlled and optimized shuffling process will enable us to mitigate a strong DDoS attack in a few rounds of shuffling. We modeled the shuffling method theoretically to determine the optimal shuffling algorithm.

We have formalized the client-to-proxy shuffling problem as an optimization problem and have developed several Greedy algorithms to solve it. We refer the reader to our papers, which describe the full details of our algorithms [48], and to Table 1 for the notations and their meanings.

Evaluation. To evaluate the optimality of the Greedy algorithm, the results of experimental implementations were compared with the theoretical upper bound of $E(N_{cu})$. Since Equation (1) is a summation of $p_j \cdot A_j$ for each individual shuffling proxy j, the maximal value of (1) cannot be greater than the sum of the max of each $p_j \cdot A_j$ when relaxing Constraint (2), i.e. $Max(E(N_{cu})) \leq K \cdot Max(p_j \cdot A_j)$. Here, $Max(p_j \cdot A_j)$ can be obtained by running subroutine *MaxProxy(*$N, 0, N-1, N_i$*)*. The comparison between the greedy algorithm and the theoretical upper bound is done via simulations under various configurations on MATLAB.

Table 1: Notations

Notation	Meaning
N	number of clients (including insiders)
N_i	number of insiders
K	number of shuffling proxies
N_{cu}	number of innocent clients to be saved
N_{ca}	number of innocent clients that are under attack
A_j	number of clients assign to the j-th proxy
p_j	probability that the j-th shuffling proxy is not under attack

We experimentally evaluated the effectiveness and overhead of the MOTAG system using experiments with a proof-of-concept prototype. First, we assessed the performance of our algorithms against simulated large-scale DDoS attacks. Next, we measured the delay incurred by the client re-assignment operations.

Simulation-based Results: To quantitatively evaluate the efficacy of our approach, we implemented both the greedy shuffling algorithm and the MLE algorithm in Matlab, and ran simulated attacks against them. We varied the number of benign clients and persistent bots across individual simulation runs to study the performance of our algorithms under different conditions. We assumed both benign clients and persistent bots arrive in a Poisson process. On average, the arrival rate of persistent bots was 5000 per 3 shuffles while that of benign clients was 100 per 3 shuffles. The shuffling algorithm decided the total number of clients assigned to each replica server. Benign clients, together with persistent bots, were randomly assigned to replica servers. Replica servers with any number of persistent bots assigned were regarded as attacked. We experimented with different, fixed numbers of shuffling replica servers performing continuous shuffling operations to save affected benign clients.

All simulations were repeated 30 times to generate data from which a mean and 99% confidence interval were calculated.

First, we ran simulations with 1000 shuffling replicas and varied numbers of benign clients and persistent bots. The results are plotted in Figure 2a. One can see that the number of shuffles required to save 80% and 95% of benign clients rises slowly with the increase in the populations of persistent bots and clients. In the worst case, a ten-fold increase in the number of persistent bots results in less than three-fold increase in the number of shuffles; and for a given number of persistent bots, a five-fold increase in the benign clients only introduces less than 70% (40) more shuffles to save the designated percentage of clients.

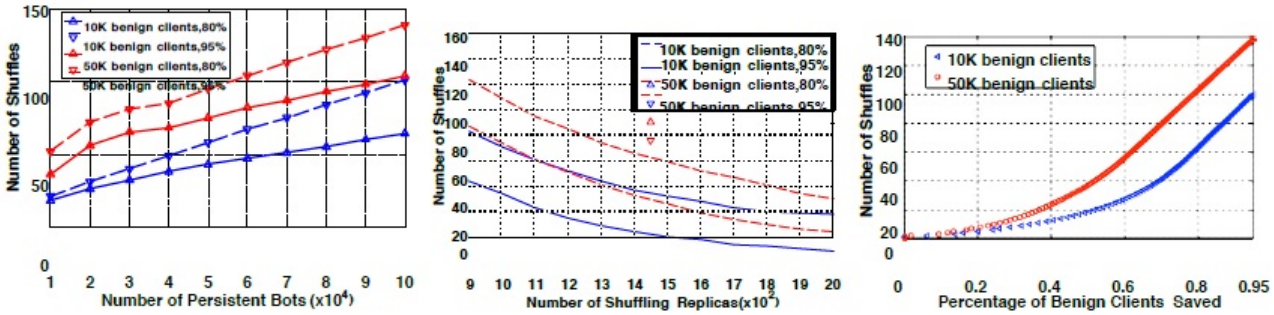


Figure 2: Shuffling Evaluation. (a) Number of shuffles to save 80% and 95% of 10^4 and 5×10^4 benign clients, with 1000 shuffling replica servers, and varying persistent bot numbers. (b) Number of shuffles to save 80% and 95% of 10^4, and 5×10^4 benign clients, with 10^5 persistent bots and varying shuffling replica server numbers. (c) Cumulative percentage of saved benign clients vs. number of shuffles, with 10^5 persistent bots, 10^4, and 5×10^4 benign clients

Next, we changed the number of shuffling replicas while keeping the client population (10^4, 5×10^4) and persistent bot population (10^5) constant. Figure 2b shows that the number of shuffles needed to save the same percentage of benign clients drops steadily when replica servers are added.

One interesting pattern consistent across both figures is that in most cases, the number of shuffles it takes to save 95% of benign clients is more than 40% higher than that to save 80% of benign clients. To explore this pattern in greater detail, we recorded the number of benign clients saved in each shuffle and plotted a cumulative percentage graph in Figure 2c. The curves show that the early shuffles were able to separate more benign clients from bots than the latter shuffles. This effect is because as more benign clients were saved, persistent bots accounted for a greater percentage of the remaining population, making it harder to separate out the benign clients.

Prototype-based Results: To study the overhead of our approach, we built a proof-of-concept prototype of the open internet service MOTAG as described by Figure 3. Shuffling for both open and authenticated clients is similar and this prototype provides insight into both approaches. We implemented two replica servers and a coordination server/coordinator on separate Amazon EC2 [1] micro instances. We used 60 PlanetLab nodes as clients. Each of the replica servers runs a simple

web server that displays a static web page (246KB) fetched from a network mounted storage. The web server logic is written in Node.js.

Initially, all clients are served by replica P_1. When a simulated attack is triggered on P_1, P_1 consults the coordinator for next step (step 1). In general, the coordinator will make shuffling decisions either by running the greedy algorithm or by looking up the pre-computed client-to- server assignment tables generated by the dynamic programming algorithm. The decisions are sent back to the attacked replica to guide subsequent shuffling operations. For the purpose of overhead evaluation, the coordinator responded by asking P_1 to redirect all clients to replica P_2 (step 2). As a result, P_1 proactively sent redirection notifications to all clients (step 3). Upon reception, clients contacted and reloaded the same web page from P_2 (step 4, 5, 6, 7).

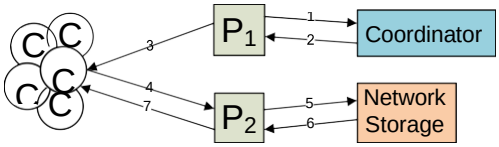


Figure 3: MOTAG Prototype -- C is client, P is replica server

Unlike conventional HTTP-based communications that start with client requests, the redirection operation is always initiated by an attacked replica server. To send unsolicited messages to HTTP clients, we take advantage of the WebSocket [38] technology multiplexing HTTP(S) ports (80 and 443). WebSocket is well-supported by all major browsers. Therefore, the adoption of our mechanism does not depend on extra software being installed on the client side. For this prototype, our server injects a snippet of JavaScript code (40 LOC) into the requested web page to establish a WebSocket between clients and the replica server.

With this prototype system, we studied the time overhead of the redirection operations associated with client-to-server shuffling. This prototype ran Firefox browsers (v17.0) on up to 60 geographically distributed PlanetLab nodes as clients, who visited the same web page (246KB) served by replica server P_1 concurrently. The time for all clients to complete steps 1-7 (i.e. redirection time from P_1 to P_2) is shown in Figure 4. In this figure, the upper curve shows the time it took to successfully redirect all clients, while the lower curve reveals the average redirection time per client. The measurement for each data point was repeated 15 times to obtain the mean and 95% confidence interval. The results show that we can re-assign 60 clients in less than 5 seconds. Overall, the data indicates a low overhead for client re-assignment operations during shuffling, but we see room for additional improvement because our server program was single-threaded and not optimized at all. The prototype was mainly developed to show the feasibility of the approach and provide a lower- bound on performance. A practical DDoS defense system with more replica servers is not expected to slow down because all replica servers act independently and in parallel.

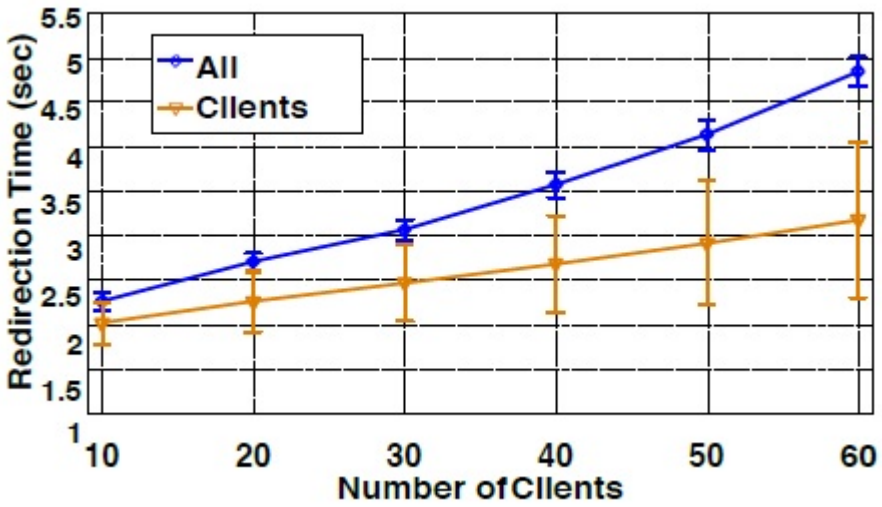


Figure 4: Client Migration Time

4.1.2 EPC: Urgent Virtual Machine Migration

Crucial to enabling a moving-target defense, such as that provided by MOTAG, is the ability to quickly migrate a running computation from one location to another. Live migration has been proposed as a way to transfer VMs efficiently and without much disruption. However an our extensive study of multiple live migration technologies (including QEMU-KVM, Xen, VMWare ESXi, and VirtualBox) indicates that known live migration mechanisms perform extremely poorly when the VM to be migration exhibits high load – as it might under a DDoS attack scenario. We have developed a new type of migration, called *urgent migration*, which is specifically suited for migration under heavy load situations, including DDoS situations [5]. Our research indicates that an effective solution for implementing urgent migration requires *enlightenment* – support built into guest operating systems (OSes) for migration. On this basis, we designed a new urgent VM migration mechanism, called *enlightened post-copy (EPC)*, which achieves fast migration of VMs under high loads while providing good performance during migration. We showed that EPC, with lightweight instrumentation inside guest Linux, successfully achieve urgent migration when existing black-box approaches fail.

Problem Statement. A decade after it was first proposed in [25], migration of virtual machines (VMs) in execution continues to be a topic of great interest and wide adoption in enterprise server consolidation [3], and more recently, in public infrastructure-as-a-service clouds [13, 40]. By allowing a VM to transparently move to a different host when its existing environment is no longer suitable, VM migration has enabled a number of compelling use-cases including scheduled maintenance of hosts with no downtime [13], dynamic load balancing [91], oversubscription of compute and memory (followed by migration when resource contention occurs) to improve infrastructure utilization, denial of service attack mitigation and quarantining [48], and nomadic computing in which cloudlet computation follows mobile users as they move from one wireless access point to another.

Much work has also been done on several variants of VM migration techniques, each with different

characteristics. They can be categorized into pre-copy migration, in which the VM's state is transferred *before* its execution is transferred to the destination (including live migration [25] and stop-and-copy [82, 95]), and post-copy [43, 44], in which state is transferred *after* execution transfer by demand fetching from the source.

However, despite all of this work, we observe that VM migration today continues to mainly be a tool for *VMs at rest*. It works best when a VM isn't doing much work, has little mutating state, and is running in lightly loaded environments with ample time for migration to occur. When a more urgent jumping of ship is called for to handle situations such as sudden resource contention in an oversubscribed system or the onset of a DDoS attack, current approaches to VM migration can fail in spectacular ways. Consider, for example, the migration of one of two VMs running heavy memcached workload on a recent version of qemu-kvm (2.3.0) and contending with each other for resources. Figure 5 shows the performance with both pre-copy (top) and post-copy (bottom) migration. Pre-copy (live) fails to migrate completely by getting stuck in an infinite loop trying to migrate ever increasing amounts of mutated state created during previous state-transfer rounds, and thus penalizes the throughput of the stationary memcached VM as well. On the other hand, post-copy migration does evacuate the migrated VM instantly and allows the stationary memcached's throughput to recover, however the migrated VM doesn't fare so well – large amounts of demand-fetching from the source host cause its throughput to be very low until the end of the experiment.

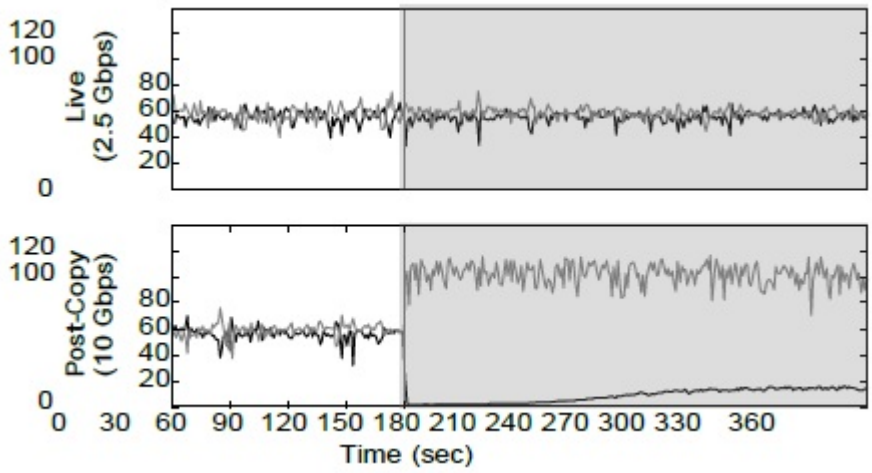


Figure 5: QEMU-KVM Live Migration vs. Post-Copy Migration

Throughput of two contending VMs (y axis) running heavy memcached workloads over time (x axis). Running individually, each VM can support 120KOps/sec; when contending, throughput is 60KOps/sec. At time=120s, migration starts for one VM (black line). The grey line corresponds to the stationary VM. With QEMU-KVM, the VM lingers on the source host indefinitely, penalizing the throughput of both VMs. With post-copy migration, VM execution is transferred instantly causing performance to recover for the stationary VM, but not the migrated one.

While it may be tempting to dismiss these issues as implementation artifacts, e.g. not terminating pre-copy after a finite number of rounds, we contend the problem is more fundamental. Any VM migration strategy is faced with a basic choice: maximize post-migration performance but

potentially delay VM evacuation by migrating state before-hand, or evacuate upfront and minimize the amount of dirty state produced during migration, but suffer post-migration performance degradation while state is fetched from the source. And the larger the VM's memory size and the busier the VM (both trends that are likely to continue as virtualized services get larger), the worse the problem. Furthermore, the right choice is very scenario-specific. While the delayed evacuation with low performance impact of live migration may be suitable for situations such as scheduled maintenance, other situations such as flash crowds, oversubscription, or DDoS attack mitigation may demand rapid evacuation to quickly remove interference for co-located VMs even at the cost of lower post-migration performance for the offending VM.

Urgent Migration. To address these tradeoffs, we make two concrete developments in our research. First, we argue that hypervisors, in addition to exporting live migration as they typically do, should also export *urgent migration* as a distinct type of migration that can be chosen by administrators for appropriate situations. Urgent migration emphasizes fast evacuation even if it is at the expense of downtime of the VM being migrated, and is semantically distinct from traditional live migration with its low downtime guarantees. Second, we show that the only current approach that could be used to implement urgent migration, post-copy, is unnecessarily limited in its performance because of its *blackbox* treatment of the migrated VM, and show how to improve its performance with a more *greybox* approach.

Specifically, we ask whether the guest operating system (OS) could provide insights that provide an opportunity for better post-copy migration. To that end, we propose *enlightened post copy* (EPC), an urgent migration mechanism that leverages cooperation between the guest and host layers in the virtualization hierarchy to improve post-migration performance. In EPC, the guest OS informs the hypervisor of memory regions that must be transferred with high priority to sustain guest performance. The hypervisor then evacuates the VM instantly just like in traditional post-copy, but subsequently pushes state to the migrated VM in prioritized order while concurrently serving demand fetches from it. We show that the priority hints can be based on information that is already easily available in the guest, e.g., whether a memory region is mapped, kernel allocated, kernel code, kernel data, user code, or page-cache.

We have implemented EPC by modifying the Linux kernel to support the creation of migration hints, and by modifying QEMU-KVM use them during migration. Through extensive experiments on memory intensive benchmark workloads such as MySQL, memcached, and Cassandra, we show that EPC is a superior solution for urgent migration compared to post-copy as well as stop-and-copy migration. Based on these results, we contend that with the widespread adoption of virtualization, embedding of lightweight yet flexible mechanisms by guest OSes to support VM migration as a first class operation is a reasonable trade-off for the improved performance. We show some results next. More details are available in our VEE paper [5].

Evaluation. We compare application-level performance of three workloads – memcached, MySQL, and Cassandra – during migration with enlightened-copy and live migration. The results assume that migration happens over a 2.5 Gbps network. Figure 6 shows the results. The y-axis shows operations per second in thousands (x1000), and total duration of migration is shown as shade areas. The dark lines indicate the performance of the migrated VM, and gray lines are that of the con-

tending VM. The source of contention is user traffic handling by the source hypervisor. With live migration (labeled Live), migration takes a long time to finish (shaded areas are long), and at least in the case of memcached (leftmost graph) it never finishes during the course of the experiment. This results in both VMs – the migrated and co-hosted VM – running at lower throughputs for a very long time. If the migrated VM is indeed under DDoS pressure, then this is a bad situation to be in. In contrast, EPC finishes migration much faster than live migration, allowing the co-hosted VM to recover its throughput very quickly after the attacked VM's migration. Memcached takes the longest time to recover for EPC, because the workload is particularly challenging for it: during this workload, almost the entire memory changes all the time, leaving little room for EPC to prioritize migration of memory pages.

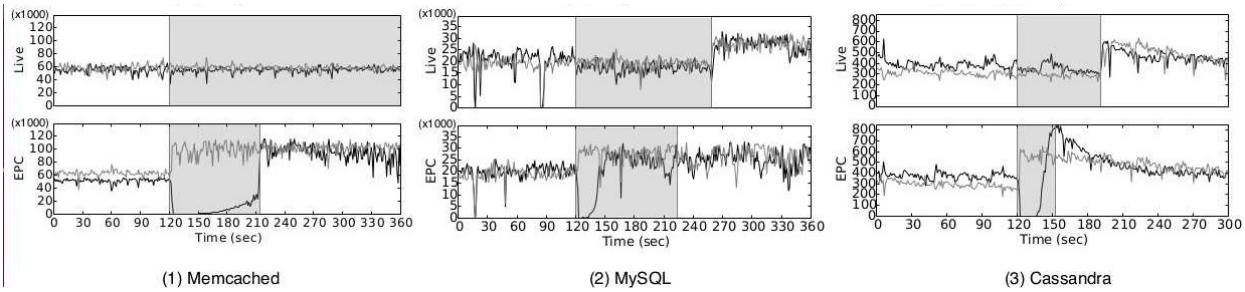


Figure 6: EPC vs. Live Migration. Results are shown for a 2.5 Gbps network

4.1.3 CleanOS (a.k.a. Evade): Continuous Data Migration

We developed an operating system (OS) that continuously cleanses itself of data. Today's OSes, including cloud OSes but also mobile OSes like Android, are extremely dirty: data gets cached at multiple layers, and deallocated or deleted data doesn't get deleted securely. All of this data is subject to compromise should the machine be compromised. To address this problem, we developed *CleanOS*, a system that monitors the use of data and "evicts" it to a key service (completely trusted) whenever it is not under active use [88]. To evict data efficiently, we encrypt it with a key that is escrowed on the trusted key service and get rid of the key from the local machine. When the data is needed again, the OS requests it from the key service. The key service keeps an audit log of all requests and evictions. If a machine is compromised, the minimal amount of information is readily available and an auditor can know exactly what data was exposed by checking.

Although CleanOS is relevant for both cloud and mobile environments, our prototype extends the Android mobile OS, hence our summary of the system's design and motivation focuses on that system. We note that in the original MEERKATS proposal the component was called Evade, but we renamed it to CleanOS, building upon and extending the same concepts such as efficient cryptographic data migration and auditing.

Problem Statement. Mobile devices are taking over as the primary computing platform for end users. Despite advantages, they are incredibly prone to theft and loss, which places all data stored on them in danger. Despite the threats, mobile OSes, like desktop and server OSes, let sensitive data accumulate uncontrollably on the device. For example, the OS accumulates significant amounts of

data in cleartext memory, and the file system retains deleted files by not purging their contents. Despite being backed by clouds, applications hoard sensitive data – such as emails, documents, and banking information – on the vulnerable device. Although encrypted file systems [70], encrypted RAM [79], and remote-wipeout systems [11, 47] help protect this data, they are imperfect stopgaps for OSes that were simply not designed with physical insecurity in mind. For example, a recent study shows that 57% of corporate users employ no locking mechanisms on their smartphones, rendering regular encryption useless [77].

Our threat model considers *any data on a mobile device to be vulnerable to data-driven thieves*. While many data protection systems exist – including encrypted file systems [70, 87, 35], encrypted RAM [79, 61, 76], and data wipeout systems [11, 47] – they are imperfect when confronted with negligent users or (sophisticated) physical attacks. First, users can foil any protection system by not locking their devices [77], assigning trivial PINs or passwords [46], or writing passwords down in easily retrievable locations [83]. Second, mobile devices are prone to physical attacks, which are notoriously difficult to protect against. For example, an attacker could use cold boot attacks [42] to retrieve in-RAM decryption keys or data, break the seal of tamper-resistant hardware [9, 81], or shield the device from the network to prevent remote wipeout [11]. Such threats are especially relevant for corporate, government, and military users, who interact with particularly sensitive data, such as trade secrets, customer data, health data, or state secrets.

To maintain post-loss control over data despite such threats, CleanOS evicts data to a key service, which is assumed to be trusted and non-compromisable. This would require incorporation of single-machine resilience mechanisms, such as the ones developed through the CRASH program. We assume that disconnection is the exception rather than the rule. In the mobile case, with pervasive wireless and cellular network coverage, this assumption is becoming increasingly realistic. In the cloud deployment case, this assumption is already true: services within a cloud are connected through high-speed networks.

CleanOS. We developed *CleanOS*, a new Android-based mobile operating system designed to *manage sensitive data rigorously and maintain a clean environment at all times in anticipation of device theft*. The crucial insight in CleanOS is to leverage the tight integration of today's mobile applications with trusted cloud-based services in order to evict sensitive in-memory and on-disk data to those services whenever it is not needed on the de- vice. CleanOS thus ensures that the minimal amount of sensitive data is exposed on the vulnerable device at any time.

CleanOS extends Android in two major ways. First, it introduces *sensitive data objects* (SDOs), a new abstraction that facilitates management of sensitive data on mobile devices. An SDO is a logical collection of Java objects, files, and database items that applications create and use to manage their sensitive data, such as emails, financial data, or documents. SDOs and their data "disappear" from the device unless they are frequently used by an application. For example, if an email app adds an email's content to an SDO, any "trace" of that content automatically disappears from RAM and stable storage unless the user is actively reading that email on an unlocked screen. Recovering the email requires interaction with the cloud.
Second, to evict idle SDOs, CleanOS modifies Android's Java interpreter (Dalvik) to introduce a new type of Java garbage collector (GC), called an *evict-idle GC* (eiGC). While a traditional GC

deallocates only those objects guaranteed to never be used in the future (i.e., no pointers to them exist), eiGC eliminates objects that have not been used for a period of time *even if* they might be used again in the future (i.e., pointers to them still exist). To do so, eiGC walks through all Java objects in an idle SDO and encrypts their data-bearing fields, such as primitives and arrays of primitives, with a key that is escrowed on a trusted key service. Our modified Dalvik interpreter then faults when a bytecode instruction executes on an evicted object, retrieves the key from the key service, and decrypts the object. Thus, data eviction in CleanOS is logical; the data itself remains on the device in encrypted form, while the key is shipped to the key service.

The major security benefit of CleanOS stems from the value-added services that the key service can build on top of it. For example, key service could revoke data access following a theft report, provide an audit log of data exposed upon theft, or monitor data access to detect anomalous uses. In a cloud setting, the key service could support post-attack forensics by providing a detailed history which specific objects were exposed on which machines at any time.

We built CleanOS in Android using the TaintDroid taint-tracking system [36] and also implemented value-added services that provide post-theft data-exposure auditing. To do so, we modified several core components in Android and TaintDroid. Together, our changes provide: (1) eviction of idle Java objects, (2) heuristics for identifying sensitive data without requiring app changes, (3) support for millions of taints in TaintDroid, and (4) multi-layer secure deallocation of freed data in Java, native, and kernel space.

Idle Data Object Eviction. Figure 7 shows the CleanOS architecture, which includes three major components: (1) the *sensitive data object* (SDO) abstraction, (2) a modified, eviction-aware version of the Dalvik interpreter, along with an *evict-idle garbage collector* (eiGC), and (3) the SDO key escrow service. Briefly, apps create SDOs and place their sensitive Java objects in them. The modified Dalvik tracks their propagation across RAM with TaintDroid and monitors their bytecodelevel accesses. The eiGC evicts SDOs to the key service if they remain idle for a specified period.

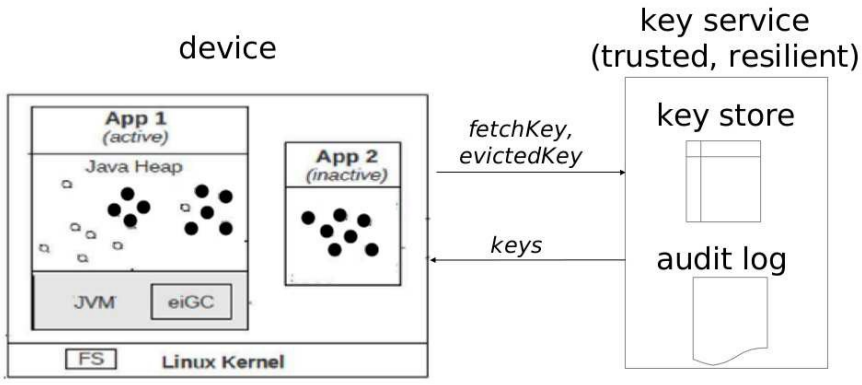


Figure 7: CleanOS Architecture

An SDO is a logical collection of Java objects, such as string objects representing the emails in a thread or objects pertaining to a bank account in a finance app. Upon creation, SDOs are assigned app-wide unique IDs and encryption keys (K_{SDO}), and are registered with the key service.

We implement three functions for SDOs. First, we track objects in an SDO with a modified TaintDroid system, using the ID as a taint. As objects are tainted with an SDO's ID, they become part of the SDO. For example, SDO1 in Figure 7 includes three objects added to it either explicitly by the app or automatically by our modified Android framework. Second, we monitor accesses to SDOs and record their timings. Whenever an app accesses an object in an SDO (e.g., to compute on it, send it over the network, or display it on screen), that SDO is marked as used. Third, we evict SDOs when they are idle for a time period (e.g., one minute).

To evict idle SDOs, the eiGC eliminates unused Java objects from RAM *even if* they are still reachable. It periodically sweeps through Java objects and evicts them if they are tainted with an idle SDO's ID. In Figure 7, the active app (App 1) has one available SDO (SDO1) and one evicted SDO (SDO2). For example, an SDO associated with an email thread might be available while the user reads emails in that thread, but the password SDO might remain evicted. When the app goes into the background, all of its SDOs might be evicted, as shown for App 2. An SDO is evicted when all Java objects in it have been evicted; however, an available SDO may have both evicted and available Java objects.

Conceptually, eviction occurs at the level of logical SDOs. In practice, however, CleanOS must eliminate the actual data-bearing objects from the vulnerable device. To do so, eiGC leverages a cryptography-based data migration and applies it to the memory subsystem. Specifically, eiGC replaces data-bearing fields in objects, such as primitives and arrays of primitives, with encrypted versions and then securely destroys the encryption key. To encrypt a data field F, eiGC uses a key K_F that is uniquely generated from the SDO's key K_{SDO} in the key service. We modified Dalvik to fault when an app attempts to access the evicted data, at which time it retrieves K_{SDO} from the key service, generates K_F, and decrypts the data. K_{SDO} is then cached onto the device and securely removed when the SDO as a whole is again evicted. This encryption-driven data eviction is significantly more efficient than shipping data to/from the key service.

We implemented an auditing service on CleanOS. Its goal is to provide users with audit logs of what was on the device at the time of theft and what has been accessed since. The auditing service integrates with the CleanOS service and both are hosted on App Engine. When a device registers an SDO or requests a decryption key, the key service logs that operation with the app name, SDO, and current time. In this way, the user can learn from the audit log exactly what data was leaked. For instance, Figure 8 shows a sample audit log that contains entries for SDO registration and key fetching. Were these operations to occur after the device was stolen, the user will know that the email password and KeePass entry may have been leaked.

device	message	time
cd5493c1befeb9075442862afa046182	fetchKey(9.408 - com.android.email - password)	2012-04-28 17:26:50.590000
cd5493c1befeb9075442862afa046182	registerSDO(com.android.email - Invitation to develop "Clean OS")	2012-04-28 17:27:01.140000
cd5493c1befeb9075442862afa046182	fetchKey(30.709 - com.android.keepass - Entry)	2012-04-28 17:27:48.500000

Figure 8: CleanOS Audit Service

Evaluation. We evaluated CleanOS's data exposure benefits and performance characteristics. Our goal was to show that CleanOS significantly reduces sensitive data exposure while providing reasonable performance even over cellular networks. We conducted all experiments on rooted Samsung Nexus S phones running CleanOS on Android 2.3.4 and TaintDroid 2.3. In this report, we focus on the data exposure evaluation.

To evaluate the data exposure benefits of CleanOS, we posed three questions: How much does eviction limit exposure of sensitive data? How much do default SDO heuristics limit exposure? How effective is the auditing service? To answer these questions, we recorded a 24-hour trace of one of the authors' phone running CleanOS as it was used to interact with regular apps, including Email, Facebook, and Mint. For Email, we experimented with both the unmodified app and our modified version of it, which we call CleanEmail. The Email app was configured with the author's personal account, which receives about ten new mails daily, and with the default 15-minute refresh period. Facebook and Mint had widgets enabled, which made them continuous services.

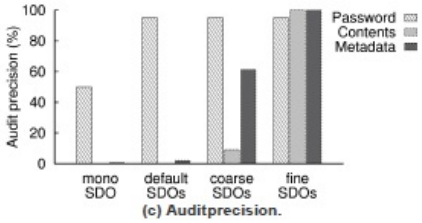
Sensitive Data Exposure Period: We measured the exposure period for three types of tainted data (password, content, and metadata) in the Email app. Figure 9(a) shows the fraction of time that each type of tainted data was exposed in RAM. Without CleanOS, the password was maintained in RAM all the time, and the content and metadata were exposed over 95% of the time. CleanOS reduced password exposure to 6.5%, which is a 93.5% reduction. For email content, the unmodified Email app with default SDOs reduced exposure time from 95.5% to 5.9%, and modifying the app

App	Password	Contents	Metadata
Email without CleanOS	100%	95.5%	99.0%
Email with default SDOs	6.5%	5.9%	5.9%
CleanEmail (fine SDOs)	6.5%	0.3%	1.6%

(a) Sensitive data exposure period.

App		Without CleanOS	With CleanOS		
			1 bucket	32 buckets	1024 buckets
Email	Password	22.5h	1h 28m	1h 19m	1h 11m
	Contents	20.9h	3 min	1 min	1 min
	Metadata	20.9h	18 min	6 min	6 min
Facebook		24h	3h 54m	3h 51m	3h 29m
Mint		24h	1h 10m	1h 2m	55 min

(b) Sensitive data lifetime.



(c) Auditprecision.

Figure 9: Data Exposure

to support fine-grained SDOs further reduced it to 0.3%. Similar observations held for metadata. To be clear, these results depend on workloads. From another, much more intensive email workload – that registered for many mailing lists and Twitter feeds – we obtained a result of 7.3% and 12.7% for content and metadata, respectively. Overall, results demonstrate a significant reduction in

exposure times for tainted data. Moreover, they show that our default heuristics protect sensitive data reasonably well.

Sensitive Data Lifetime: As SDO lifetime is critical to system security, we must also examine the maximum period that a tainted object could be retained in RAM. Figure 9(b) shows the retention time for the longest-lived tainted object in three applications, where we break down email into three types. Without CleanOS, all observed applications retained certain tainted objects for more than 20 hours. With CleanOS, the maximum SDO lifetime was dramatically reduced. For instance, the Email app kept some metadata objects for as long as 20.9 hours, which CleanOS reduced to only 6 minutes when using 1024 buckets. For Facebook and Mint, the impact of bucketing on sensitive data lifetime was more limited because these apps tend to use most objects in an SDO at the same time. Overall, these results indicated that the mobile device was significantly cleaner with CleanOS.

Audit Precision: We next evaluated the effectiveness of the auditing service we built on CleanOS (see Figure 8). We compared audit precision across four levels of SDO granularity in Email: (1) mono-SDO, where we marked data as only "sensitive" or "non-sensitive," (2) default SDOs, where we used default heuristics, (3) coarse SDOs, where the application defined one content SDO and one metadata SDO for all emails, and (4) fine SDOs, where each email had its own content and metadata SDOs. We define *audit precision* as the average probability over time that the tainted data is actually exposed on the device, given that the audit log shows its SDO has not been evicted. Figure 9(c) shows audit precision for the Email app's password, content, and metadata. Password auditing was 50.0% precise with mono-SDO but increased to 95.1% with default SDOs. The content and metadata, however, had poor precision (<3%) without application support: CleanOS could not differentiate data coming from the Internet and hence added every incoming object to the *SSL* SDO. With coarse, application-specific SDOs, audit precision for email content and metadata was 9.1% and 61.3%, respectively. When fine application-specific SDOs were available, audit precision reached 100%. Thus, our default SDOs were effective in auditing password exposure, but application adaptation was needed to provide precise auditing for other types of sensitive data.

4.2 Cloud Information Flow Tracking Technologies

As a key component of a resilient cloud, we have developed the first comprehensive framework for tracking information flows in large-scale computing clouds. This corresponds to our CIFT component of the MEERKATS proposal. Our framework consists of three, integrated components:

1. We developed *CloudFence* [74], the first cloud-wide information flow tracking system; it provides transparent, fine-grained data tracking capabilities to both service providers, as well as their users.

2. We developed *Cloudopsy* [100], an information flow visualization system that offers a visual autopsy of the exchange of user data on cloud premises. We have successfully integrated CloudFence and Cloudopsy.

3. We developed *data provenance framework* [39] which gathers data provenance information for real-world applications without any code modifications.

4.2.1 CloudFence: Data Flow Tracking for Clouds

Most feature-rich cloud-based services are quite complex, and are usually built by "gluing" together a multitude of components. Bugs and vulnerabilities in existing code, misconfigurations and incorrect assumptions about the interaction between different components, or even simple causes like the careless handling of access credentials, can lead to the accidental exposure of critical data or leave the system vulnerable to data theft. At the same time, cloud computing encourages rapid application deployment, and time-to-market pressure sometimes makes data safety a secondary priority.

To reinforce the confidence of end users for the safety of their data, we have developed *CloudFence* [74], a new data flow tracking framework for cloud-based applications, which lets users audit how their cloud-resident data flows through cloud-based services. CloudFence can be offered by cloud hosting providers as a service to their tenants, as well as to the users of the tenants' services. Through a simple API, service providers can easily integrate data flow tracking in their services and mark sensitive user data that needs to be protected. End users can then monitor the propagation of their data directly through the cloud hosting provider, ensure that all sensitive data is treated as expected, and spot any deviations. Service providers can also take advantage of data flow tracking for enabling an additional layer of protection against data leaks, by preventing the propagation of marked data beyond a set of specified network and file system locations, as well as for protecting their own digital assets (e.g., configuration files or back-end databases).

CloudFence Architecture. Figure 10 shows the main interactions among the different parties that are involved in CloudFence-enabled services. Users register with the cloud provider (1), and then use the services offered by various service providers using the same set of credentials (2). Sensitive data are tagged and tracked transparently throughout the cloud infrastructure (3). Users can audit their data through a web interface exposed directly by the cloud provider (4). To facilitate the monitoring of user data, end users have also access to the Cloudopsy dashboard with meaningful log messages and a visual representation of the audit trails of their data.

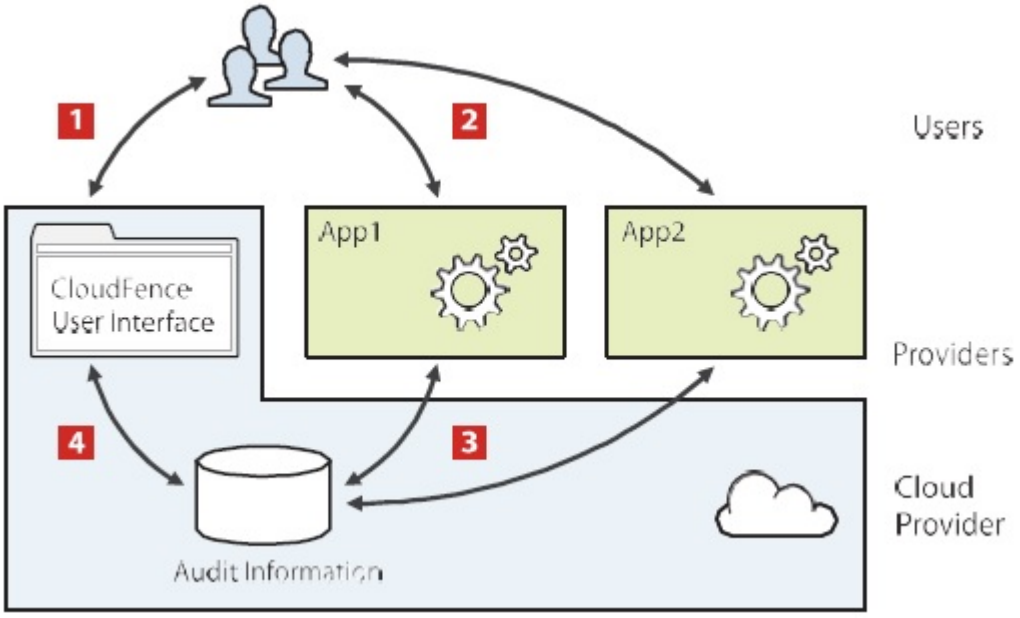


Figure 10: CloudFence Architecture

A major challenge in supporting data auditing for services with a very large number of users is the need for concurrent propagation of tagged data that carry different tags for each user. At the same time, data tracking must be performed at a fine-grained level to allow for precise tracking of user data as small as a credit card number. CloudFence introduces a novel data flow tracking framework based on runtime binary instrumentation that supports byte-level data tagging, and 32-bit wide tags per byte, enabling fine-grained data tracking for up to four billion users. Cross-application and cross-host tag propagation is handled transparently, without requiring any modifications to application code. Despite the significant increase in tag space, the runtime overhead of CloudFence is comparable to existing byte-level data flow tracking systems that support just a single or up to eight tags, and an order of magnitude lower compared to systems that support arbitrarily many tags.

Evaluation. We have successfully evaluated the effectiveness of CloudFence using two real-world applications, and two publicly disclosed data leakage vulnerabilities in those applications. Cloud-Fence can be easily integrated in both applications through the placement of just a few API calls, while it offers effective protection against a wide range of data theft threats, including SQL injection and arbitrary file read attacks.

To assess the runtime overhead of CloudFence we compare it against Libdft [51], a data flow tracking framework for commodity systems, as well as the unmodified application in each case. Surprisingly, CloudFence not only provides extra functionality that is crucial for cloud environments, but at the same time it is even faster than Libdft for the cases we considered. The extreme case in which each add operation generates a new tag resulting in a 20X slowdown (upper bound).

4.2.2 Cloudopsy: Visualization Tool for Cloud Data Flow

We have developed a visualization tool for information flows, which we integrated with Cloud-Fence. Our tool, *Cloudopsy* [100], is a service that provides secure and comprehensible data auditing capabilities to both the service providers and their clients for user data collected in the realm of these cloud-hosted services. Cloudopsy is particularly useful for users of cloud infrastructures who wish to audit how their data is being used by third-parties interacting with that data in the context of the same cloud infrastructure (e.g., service co-resident on Amazon AWS). Even though information flow auditing and enforcement techniques use similar mechanisms, we focused our efforts in auditing, due to the concerns that enforcement could have adverse effects like breaking parts of services.

Cloudopsy's mechanism is divided in three main components: a) the generation of the audit trails, b) the transformation of the audit logs for efficient processing, and c) the visualization of the resulting audit trails. Cloudopsy operates on together with our cloud-wide data tracking framework (see upcoming Subsection) and it enhances with the power of visualization, when presenting the final audit information to data owners and service providers. Notably, this novel feature of Cloudopsy significantly reinforces end-user awareness regarding the use and exposure of their sensitive data by the cloud-hosted applications.

We have integrated Cloudopsy with CloudFence, thus making Cloudopsy the user interface of CloudFence. Figure 11 displays the movement of a single user's "marked" sensitive data, i.e., credit card number and email address, as they move in time between the audit-enabled applications.

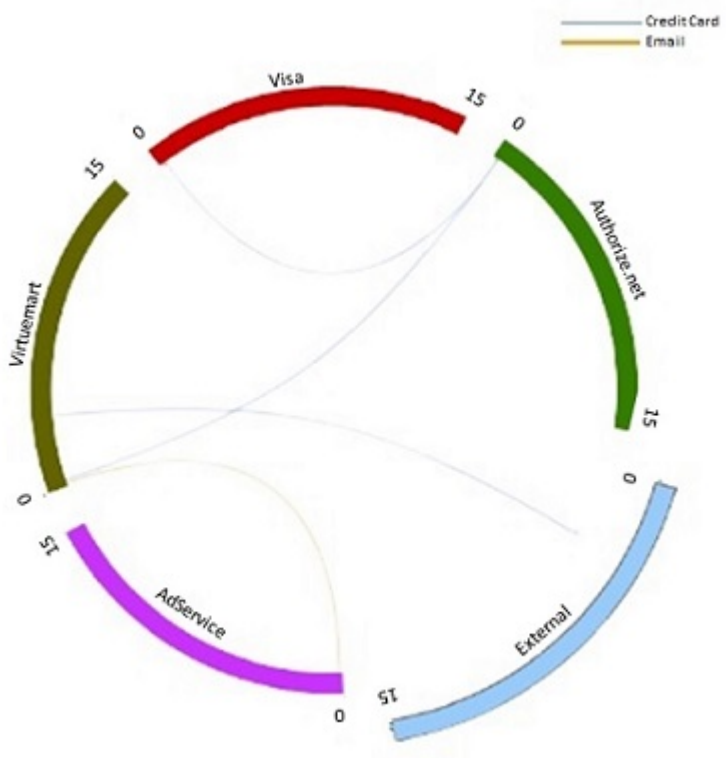


Figure 11: Cloudosy Screenshot

4.2.3 Data Provenance with Dynamic Instrumentation

Data provenance – describing how an object came to be in its present state – is an area that has drawn much research attention lately. There is a number of schemes that have been proposed on how to apply it in practice and in widely used applications. Some representative provenance schemes have been proposed for databases [24], web applications [56], and notably cloud computing [72]. One could argue that applying data provenance at a low level, e.g. at the system call-level, would be sufficient. However, that is not always the case. Applying data provenance to different layers of the software stack can provide different levels of information. For instance, in the case of a web browser case, data provenance provided solely at the system-call level would be unable to capture and record higher-level information. This can be the URLs the browser renders, in the case the communication channel is over HTTPS. On the other hand, the function-call level, as provided by OpenSSL, has access to the encrypted data stream used by an HTTPS connection.

Based on the example above, one could also argue that data provenance must be provided at the layer closest to the information we would like to capture and document. This may prove to be a very challenging task, especially when dealing with large and complex software applications such as web browsers. For example, Firefox 4 consists of more than 5M lines of code, spread in almost 40,000 files.

Data Provenance Framework. Instead of exploring how we can extend an application to provide data provenance, we took an alternate route: we proposed, and built, a universal data provenance framework using dynamic instrumentation. The main goal of our framework was to provide an easy way to prototype any data provenance scheme, no matter the size and complexity of the sys- tem it is applied on. We argue that lightening the implementation burden of such data provenance applications would lead research in this field forward, allowing researchers to test and evaluate their ideas more easily. Our framework was built on dynamic instrumentation, and especially on DTrace. Its key feature is that it can greatly assist the user in discovering paths in the system that interesting data pass through. For example, all URLs and search keywords that flow inside a browser. After the discovery phase, the user can dynamically instrument these points to record provenance about this transit data. Moreover, in cases where the source code of a system is available, our framework could be used for simply discovering points of interest in the system and evaluating a data provenance application. Then, prototyping a low-overhead source-code level implementation of the data provenance application at production level is trivial, as the user knows exactly which points to instrument in the system.

We need to mention here that the choice of dynamic instrumentation for such cases seems ideal as it provides several advantages. First of all, it requires no changes to the source code of the system that data provenance capabilities are going to be added in. Second, it can be easily enabled or disabled, even at runtime in some cases, as we shall see later on. Finally, it removes the requirement of having the source code, although having it could be helpful during the discovery phase. On the other hand, the main disadvantage of dynamic instrumentation is its runtime overhead. However, this is not an issue in our case as we perform system-call and function-level, but not instruction-level

instrumentation, which can be extremely expensive.

Evaluation. We have demonstrated the functionality of out framework in three case studies with real world systems, namely: a file system, a database (SQLite), and a web browser (Safari). The last usage scenario, extracting provenance information from a web browser, revealed some interesting limitations of our framework against very large and complex systems. These limitations are mostly due to the restricted instrumentation actions of DTrace. Combining a more powerful tool in our framework, like the dynamic binary instrumentation of Pin, seems sufficient to overcome these limitations, but in some extra cost. Also, tools implemented on top of Pin, like CloudFence, could potentially add more features to our framework.

4.3 Misinformation and Decoy Technologies

Our MEERKATS resilient cloud vision included a critical component, called *DIGIT*, capable of generating, injecting, and tracking deceptive information aimed at detecting and confusing adversaries. As part of DIGIT, we developed and commercialized several systems:

1. *DIGIT* introduces *computational decoys*, where realistically-looking service requests are automatically manufactured and introduced into the flow of a system [54].
2. *Novo* introduces *document decoys*, where realistically-looking documents are manufactured and instrumented to catch device thieves. Novo is a commercial product that is being tested at various companies and government agencies.
3. *SAuth* introduces *password decoys*, where fake passwords that are indistinguishable from real user passwords are automatically inserted into a service's password database in anticipation of whole-database password leaks [53].

4.3.1 DIGIT: Computational Decoys

We have developed the concept of computational decoys, a novel approach that encompasses deceptive information and "throw-away" computation to impede the ability of an adversary to take advantage of any initial success they may have in compromising a system. The main goal of our approach is to introduce uncertainty as to the validity and authenticity of data captured by an adversary after gaining unauthorized access in one or more hosts, and in some scenarios, reveal the presence of the adversary.

DIGIT. We have applied this approach in DIGIT [54], a deceptive information generation, injection, and tracking system aimed at detecting and confusing adversaries in cloud settings. The system is based on a large number of application replicas, some of which (the "deception set") are provided with fake inputs. An adversary controlling a malicious or compromised replica will be uncertain as to the validity of any captured data. The whole process is orchestrated by an application level proxy that mixes and dispatches real and fake requests to primary and decoy application replicas, respectively. Primary and decoy replicas can be swapped at any time simply by changing the source of inputs, increasing the confusion to potential adversaries. The focus of our prototype is the

automated generation of HTTP message decoys.

Evaluation. To evaluate our prototype we have implemented and experimented with a simple web service over HTTP. The service exposes a simple search engine interface that accepts keyword searches. By dynamically instrumenting the application server, we were able to focus on application behavior and capture control flow properties and execution operations. In our test scenario we performed various keyword lookups in the test search engine service. Measurable characteristics included the outcome of the query, the time to process a request, and the response size– all properties directly affected by user input. Decoy inputs were organized into labeled groups based on the behavior of the instrumented server (short/long response time, fast/slow processing, valid/invalid response). For our evaluation we used an actual trace of user search engine queries (single word input) to evaluate the abstracted decoys, and monitored the behavior of the application given (a) a sample from the actual trace, (b) a sample from the generated decoys, and (c) a random mix of the two samples. An interesting observation was that fake information, when processed by an application, can result in dramatically different code execution paths and overall behavioral side effects. For instance, if an ill-formatted HTTP request is provided, the application will likely exit with error, revealing that the request was a fake.

4.3.2 Novo: Document and App Decoys (Commercialized)

A variant of the DIGIT deception technology was transitioned under DARPA support to Allure Security Technology for incorporation in Allure's product, Novo. Novo's focuses on unusual user behavior and detection of the touching or exfiltration of strategically placed deceptive material. These include decoy documents and other bogus but believable media such as decoy software, decoy cloud services, and decoy apps in the mobile case. The DIGIT component was designed and prototyped to rapidly generate and place decoy materials as a cloud based security service. Novo incorporates this functionality in response to detected attack, whether in a cloud service, host, or mobile phone. The DIGIT technology incorporated in Novo can rapidly inject a decoy file system on a mobile phone or a laptop or desktop. The injection process includes pre-positioned cloud storage services such as Box and Dropbox containing highly believable decoy documents in Office and PDF formats. Novo is soon to be released as a GA product and is being demonstrated to various departments of the DoD who are interested in deploying Allure's Novo decoy technology.

4.3.3 SAuth: Password Decoys

Password theft can cause annoyance, financial damages, and loss of privacy. Services employing passwords urge their users to choose complex combinations, change them frequently, and verify the authenticity of the site before logging in. However, even users that manage to follow all these rules risk having their passwords stolen. Security pitfalls in a series of popular services have resulted in frequent and massive password leaks. Even though services usually store a digest or hash of the password (excluding rare incidents), the emergence of powerful password-cracking platforms has enabled attackers to recover the original passwords in an efficient manner. What is more, password-reuse practices by the users have enabled domino-like attacks

We have developed Synergetic Authentication (SAuth), an authentication mechanism based on the

synergy between different services that complements their individual procedures for verifying the identity of a given user. To successfully log into service S one is required to successfully authenticate both with S and a cooperating vouching service V. For example, a user logging into his Gmail account, after successfully submitting his password to the Gmail server, he will be required to also submit his Facebook password to the Facebook server. Once Gmail receives notice from Facebook that the same user has managed to authenticate successfully it will have an additional assurance that the user is the actual owner of those accounts. The approach is founded upon the way most users access the web. In particular, users remain concurrently and constantly authenticated with many services such as e-mail and social networks unless they explicitly log out. A service the user is already logged on can transparently vouch for him, e.g., through client-side cookies. In the above example, had the user been already logged in to Facebook, he would just be required to enter his Gmail password to access his e-mails. Facebook would use his user agent cookies to transparently authenticate him and subsequently vouch for him to Gmail. At the same time, an attacker that has compromised the user's password for S is unable to access that account as he is lacking the password for vouching service V and thus cannot complete the authentication process. In other words, for an attacker to compromise one account, he must acquire multiple account passwords for that user located in different databases of distinct services.

Problem Statement. Password-based authentication has received a lot of criticism lately with many large services like Google and PayPal looking for alternative means to authenticate users. Some alternatives that have been proposed in the past include public-key mechanisms, such as TLS client certificates and more. Unfortunately, none of the proposed alternatives has proven sufficiently enticing. Passwords have been the de facto method for authenticating users for many decades, and have proven to be resilient to change.

Two-factor authentication [6] has probably been the most successful proposal to complement password-based systems by requiring that an additional password is provided, acquired through a second independent channel. Unfortunately the overhead both in cost and effort to deploy and maintain such system has led to adoption only by high-value services such as banking sites and e-mail providers. Moreover, it scales poorly when users are required to manage multiple secondary factors for distinct services. Finally, a study has shown that it can push users to weaker passwords. Single-sign-on services like OpenID, as well as the OAuth-based interfaces of social networking services, offer the alternative of maintaining a single on-line identity, protected by one, hopefully strong, password. Users of such services, instead of creating separate, new accounts and passwords with third-party services, authenticate with a trusted identity provider (e.g., Facebook), that vouches for them. However, these identity providers present a single point of failure, may carry privacy-related risks, and can also suffer vulnerabilities themselves. A recent study attributes the limited adoption of such services to concerns regarding their availability and relinquishing control of the user base as part of outsourcing authentication.

SAuth Architecture. SAuth does not suffer from any of the above problems, as it complements, rather than substitutes, existing authentication procedures at each site. Therefore, it does not degrade the security of a service, but allows heterogeneous authentication systems to operate at each site, while preserving each party's user accounting system. Finally, it encourages symmetric relations between services as it enables all participating sites to operate both as relying and

vouching parties if they wish to. In Figure 12, we provide the steps executed for Alice to successfully authenticate with service S in the presence of service V, which is trusted by S and has agreed to vouch for its own users to V.

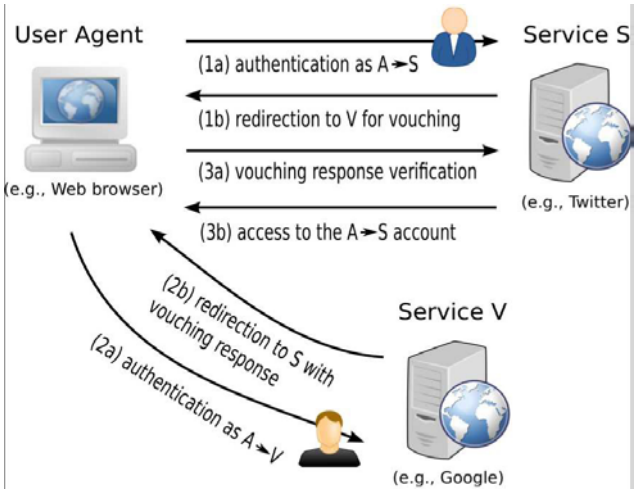


Figure 12: Synergy-Enhanced Authentication

The steps are as follows. A user, with accounts in services S and V (e.g., Twitter and Google) tries to log in S. A standard authentication process takes place which involves the user's password for S (step 1a). Subsequently service S initiates our proposed protocol which involves service V vouching for the user (step 1b). The user is redirected to V with which he also engages in a standard authentication process (step 2a) at the end of which service V generates a vouching token for the user to return to S (step 2b). Finally, service S allows the user to access his account if and only if the vouching token from V is verified (step 3a) and optionally returns a persistent authentication token (e.g., cookie) that will bypass this authentication process for subsequent interactions (e.g., HTTP requests) with S.

To address the issue of password reuse, which can render vouching ineffective if a user recycles the same password across all vouching services, we employed decoy passwords. Each service automatically generates multiple decoy passwords that are similar to the one chosen by the user. Furthermore, for the decoys to blend in, they receive no special treatment by the service and are thus considered as legitimate credentials for user authentication. Note however, that the user is never aware of them. Enabling decoy passwords requires no changes in the database schema, can be implemented with wrappers to existing password management functions and introduces multiple password entries per user instead of a single one. An attacker cracking the passwords will be unable to identify the actual one and resort to on-line guessing against the vouching services.

Evaluation. We have extensively evaluated the security of SAuth and compare it with the current practice of single-site authentication. In particular we have considered four cases: The first refers to the normal scenario where someone without any prior knowledge of the password attempts to log in to a service. The second is a special sub-case where the user of the target account is reusing his password in more than one services. The third and the fourth, refer to the scenario where a service

has been compromised and the user's password for that service leaked. Our results indicated that our approach offers a significant increase in the security of all cases except for the fourth.

4.4 Cloud Monitoring and Self-Healing Technologies

A fourth class of critical components of a resilient class are continuous monitoring systems, which can detect emerging attacks before they inflict damage and either defeat them through self-healing or direct other defense components, such as migration or decoy systems, to start defending the system. In this realm, we developed several systems, which we next describe.

4.4.1 DMCC: Distributed Monitoring and Cross Checking

Distributed Monitoring and Cross Checking (DMCC) is a core subsystem of the MEERKATS cloud resilience vision. It performs distributed monitoring and cross checking to detect any attacks against cloud services and to trigger mitigation actions, such as data migration and decoy services. This is mainly accomplished through the correlation of alerts generated by anomaly detection (AD) sensors placed at different layers of abstractions in the system. We achieved our project goal in a few sub-tasks.

We developed three approaches to perform alerts correlation in DMCC [94, 101]. The first approach is called cross-site correlation, where we proposed and evaluated a distributed model aggregation and exchange scheme to solve dynamic path problem introduced by load balancing in the cloud environment. We then investigate cross-layer and cross-sensor correlations to implement defense in depth and breadth using multiple security controls. Cross-sensor alerts correlation aggregates alerts from multiple security controls deployed at the same execution layer, and cross-layer alerts correlation integrates alerts from sensors placed across multiple execution layers aligned with attack vectors. Together they also provide an effective unified defense for any attack mitigation mechanism and helps to reconstruct attack scenarios for forensic purposes.

To facilitate system monitoring in DMCC and the integration with other MEERKATS components, we have designed and built a security architecture benchmark, called the Wind Tunnel system [19], that performs synthetic data generation and multi-layered alerts correlation. We experiment with nineteen security controls across three pre-defined layers - network, database and host, to demonstrate the capability and usability of Wind Tunnel system. In our experiments, we focus on server-side web applications with real-world web attacks that exhibit multistage components. We implement three classic correlation functions and invent three similarity based correlation functions, and study their performance empirically using Wind Tunnel system. We quantitatively demonstrate the improvement on detection performance while minimizing false positives with a systematic comparison of several correlation techniques.

Evaluation. We built the Wind Tunnel benchmark with nineteen supported security controls across three execution layers with six alerts correlation methods. Researchers interested in using Wind Tunnel to generate data and/or evaluate web application defenses are welcome to contact us. We have two virtual machines with instructions to deploy Wind Tunnel as well as the source code available for noncommercial research purposes.

4.4.2 CRP: Self-Healing Multitier Defense

To mitigate the effects of security bugs that can reduce the integrity of systems, a plethora of run-time protection mechanisms have been devised. Nevertheless, while protection mechanisms render certain types of vulnerabilities infeasible or impractical, they do not also offer high availability and reliability, as they frequently resort to terminating applications that behave abnormally to prevent attackers from performing any useful action.

To increase software availability, many mechanisms that aim to recover execution when unhandled errors occur have been proposed. One of these mechanisms is software self-healing based on rescue points [84]. It operates based on the observation that applications already contain code for handling anticipated errors and proposes reusing this code to also handle unexpected errors. Rescue points (RPs) are essentially functions that contain error handling code, which can be exploited to recover from errors occurring within the RP, including the RP routine itself and all called routines. A checkpoint is taken upon entering a RP, and execution is rolled back to that checkpoint when an unhandled error occurs, while concurrently a valid error code is returned by the RP to the application (i.e., through the routine's return value), so that it can gracefully handle the failure.

Applying RP-based self-healing on self-contained functions is straightforward; however there are many functions that have side effects, such as transmitting data to other entities on the network. Applications that are part of multi-tier architectures, like client-server or three-tier architectures (comprised by presentation, logic, and data tiers), contain many such functions. Introducing RPs in such architectures can be problematic because it can result in inconsistent states between the tiers when a roll back occurs.

We have developed cascading rescue points (CRPs) for self-healing applications in multi-tier architectures to address the inconsistency issues introduced by traditional RPs. In our approach, when an application executing within a RP communicates with an application on the next tier, we notify the remote peer to also perform a checkpoint, cascading, in this way, the checkpoint and RP to the lower tiers of the architecture. If a RP successfully completes execution or if it triggers a roll back due to an error occurring, a notification is also send to all the peers that were instructed to checkpoint, so that they also perform the appropriate action.

We have implemented CRPs using the Pin [64] dynamic binary instrumentation framework for x86 Linux, extending our previous work [78] on deploying traditional RPs using Pin. We improve the check pointing mechanism used by utilizing the fork system call to quickly create copy-on-write copies of an application's image and use filters to mark the individual bytes modified by threads for efficient thread-wide check-pointing. We also intercept system calls to restore the contents of overwritten memory and to transparently inject information in the communication channels between applications of different tiers that run on top of our tool. We use the injected data to implement a protocol for conveying notifications between the various parties. Additionally, we utilize TCP out-of-band data to asynchronously notify remote peers of a successful exit from a RP.

Evaluation. We have evaluated CRPs using popular server applications, like Apache and MySQL, which suffer from well-known vulnerabilities and show that our CRP protocol does not introduce prohibitive overheads. The performance overhead imposed by CRP varies between 4.54% and 71.96% depending on the application. Note that CRP can be ported with moderate effort to operate on other platforms supported by Pin, including Windows and BSD operating systems, and the x86-64 architecture.

4.4.3 Virtual Application Partitioning

Software faces a multitude of threats that enable attackers to execute arbitrary code through code-injection and code-reuse attacks, bypass security mechanisms like user authentication, and exfiltrate sensitive data. Oftentimes, attacks are enabled by bad system design and configuration errors, or even originate from otherwise "trusted" users. A plethora of defensive mechanisms that mitigate such threats have been proposed in the past like memory allocators [7] and data-flow integrity [22], but very few of them have seen broad adoption.

Two of the largest obstacles in the adoption of these security defenses are the requirement for source code by approaches that are applied at compile time and the significant performance overhead imposed by solutions operating solely on binaries. Techniques that are both lightweight and require no software recompilation have been also proposed, but they often address a narrower set of threats. Furthermore, many of these techniques are not orthogonal to each other and cannot be easily integrated. That is, even when they can be combined, their cumulative overhead becomes prohibitive.

We have observed that software does not have uniform security requirements throughout its execution. In particular, different parts or components of an application are usually targeted by different types of attacks, or suffer dissimilar exposure to a specific threat, because of external events or innate properties of the software. For instance, protecting a service against sensitive data leaks is only relevant after first reading the data. Similarly, consider an FTP server that serves both authorized and public users. In the latter case, the server is intrinsically more exposed to exploits because more of its functionality is accessible by everyone. In both examples, execution is partitioned to segments with different security requirements. If we can identify these partitions, we can apply diverse protection mechanisms as and when needed to control their exposure to different types of threats, while avoiding cumulative overheads.

We have proposed virtually partitioning the execution of applications, and adapting the defenses being deployed based on the executing partition. The benefits of using adaptive defenses are twofold. First, it enables us to apply multiple protection mechanisms selectively, disabling mechanisms that are not relevant, or of a high priority, in favor of deploying more appropriate ones. Second, it provides a risk management mechanism that is orthogonal to existing protection schemes. In essence, virtual partitioning provides a tunable knob that controls the intensity of the defenses being applied in exchange for more resources (e.g., CPU cycles or memory).

Our work focused on automatically identifying how intrinsic properties of the software partition it to segments with disparate security requirements. We postulated that using events, such as reading

sensitive data from a database, for separating an application into different partitions is straightforward and can in fact be implemented by intercepting process-operating system interactions. We supported this claim by implementing a runtime environment that can partition applications based on such events, and apply information flow tracking to monitor sensitive information.

However, our main goal was to develop a more generic and flexible framework for virtually partitioning applications. We also introduced a methodology for automatically determining how and where user authentication splits application execution to partitions with asymmetric exposure to attacks, by dynamically profiling binaries at runtime and without the need for source code, debugging symbols, or any other information about the application at hand. We have developed a runtime environment that uses the virtual partitions to dynamically adapt the defensive mechanisms applied on binaries at runtime. We reuse existing schemes, such as dynamic taint analysis and instruction- set randomization (ISR) [50], and apply them selectively on the pre and post-authentication parts of the application.

Evaluation. We have applied our approach on well-known server applications utilizing their built-in authentication mechanisms, as well as commonly used frameworks such as pluggable authentication modules. Our findings demonstrated that we can automatically identify their authentication points with accuracy. We have also evaluated our runtime to verify its ability to dynamically switch between different protection mechanisms, and determine the performance benefits that can be gained by reducing the intensity of defenses after user authentication. For this purpose, we utilized dynamic taint analysis and ISR on the pre and post-authentication partitions respectively. In addition, our results showed that we can greatly reduce the user-observable overhead of dynamic taint analysis by up to 5X, by replacing dynamic taint analysis with ISR after the user successfully authenticates.

4.5 Stable Multithreading Technologies

A fifth component of the MEERKATS cloud resilience vision deals with the significant risks posed by multi-threaded programs. These programs are notoriously difficult to write correctly and safely, as well as to test and debug. Unfortunately, bugs in multithreaded programs can lead to serious vulnerabilities such as time of check time of use attacks and memory corruptions, which attackers can exploit to gain unauthorized information, corrupt critical data structures, and execute code. Hence, a key part of our MEERKATS effort has been to develop new approaches for safer and more reliable multithreaded programming. In this realm, we made two key contributions:

1. *Stable Multithreading* (StableMT) is a new multithreading paradigm, in which a multithreaded program tries to take on each execution a schedule that is known to be safe [98]. We developed a number of systems to implement and improve stable multithreading, including our latest system, Parrot [29].

2. *Crane, a stable replication system* that leverages StableMT and Parrot to enable for the first time the transparent replication of arbitrary non-deterministic programs [27].

Both systems have been released opensource on GitHub.

4.5.1 Stable Multithreading

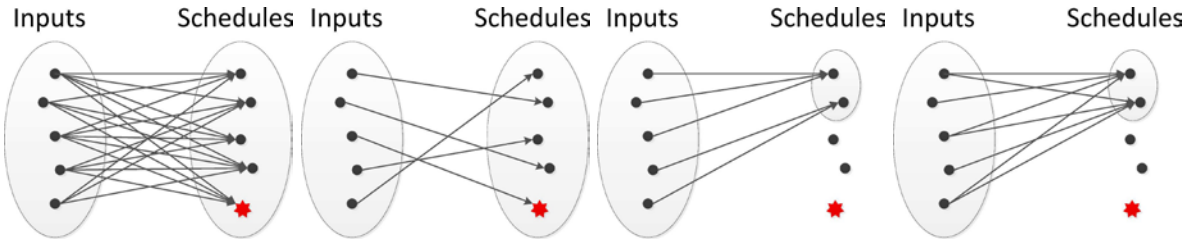
Multithreaded programs have become pervasive due to the accelerating computational demand and the rise of multi-core hardware. Unfortunately, despite much research and engineering effort, these programs are still notoriously difficult to get right. They are plagued with concurrency bugs, which can easily cause severe program behaviors such as wrong outputs, program crashes, and security breaches.

Our research [98] reveals that a key reason of this difficulty is that multithreaded programs have too many possible thread interleavings, or schedules. Even given only a single input, a program may run into excessive schedules, depending on factors such as hardware timing and OS scheduling. Considering all inputs, the number of schedules is even much greater. Finding a buggy schedule in this huge schedule set is like finding a needle in a haystack, which aggravates understanding, testing, and analyzing of programs. For instance, testing is ineffective because the schedules tested in the lab may not be the ones run in the field.

To reduce the number of schedules for all inputs, we have studied the relation between inputs and schedules of real-world programs, and made a surprising discovery: many programs require only a small set of schedules to efficiently process a wide range of inputs. Leveraging this discovery, we have proposed the idea of stable multithreading (StableMT) [98, 29] that reuses each schedule on a wide range of inputs. StableMT conceptually maps all inputs to a greatly reduced set of schedules, drastically shrinking the "haystack," making the "needles" much easier to find. StableMT can greatly benefit understanding multithreaded programs and many reliability techniques, including testing, debugging, replication, and verification. For instance, testing schedules in such a much smaller schedule set becomes a lot more effective.

StableMT is complementary to another idea called deterministic multi-threading (DMT) that enforces the same schedule on each input. Our research [98] has shown that although DMT is useful, it is not as useful as commonly perceived, because a DMT system can map slightly changed inputs to very different schedules, which can easily cause very different program behaviors. Furthermore, the number of schedules for all inputs in DMT can still be extremely large, which is the key problem that StableMT mitigates.

Figure 13 shows different multithreading approaches in a conceptual level. Traditional



(a) *Traditional.* (b) *Deterministic.* (c) *Stable (deterministic).* (d) *Stable (nondeterministic)*

Figure 13: Multithreading Alternatives - Red stars represent buggy schedules

multithreading (Figure 13a) is a conceptual many-to-many mapping between inputs and schedules. DMT (Figure 13b) may map each input to an arbitrary schedule, reducing programs' robustness on input perturbations. StableMT (Figure 13c and Figure 13d) reduces the total set of schedules for all inputs (represented by the shrunk ellipses), increasing robustness and improving reliability. StableMT is complementary to DMT: a StableMT system can be deterministic (Figure 13c) or non-deterministic (Figure 13d). Notably, StableMT has attracted the research community's interest since it was initially proposed by us [31], and most subsequent systems are both StableMT and DMT.

To realize StableMT, we have built three systems, Tern [31], Peregrine [30], and Parrot [29], each addressing a distinct challenge. In particular, Parrot, our most mature StableMT system, provides an efficient and easy to use stable multithreading runtime and is available open source at github.com/columbia/parrot. Moreover, to justify the benefits of StableMT, we have applied StableMT to several reliability and security topics, including improving precision of static analysis [96], detecting security rule violations [28], and building transparent state machine replication service for general multithreaded programs [27]. We discuss the last topic next.

4.5.2 Crane: Correctly Replicating Nondeterministic Executions

State machine replication (SMR) leverages distributed consensus protocols such as Paxos to keep multiple replicas of a program consistent in face of replica failures or network partitions. This fault tolerance is enticing on implementing a principled SMR system that replicates general programs, especially server programs that demand high availability. Unfortunately, SMR assumes deterministic execution, but most server programs are multithreaded and thus nondeterministic. Moreover, existing SMR systems provide narrow state machine interfaces to suit specific programs, and it can be quite strenuous and error-prone to orchestrate a general program into these interfaces.

We have built Crane, an SMR system that transparently replicates general server programs. Crane achieves distributed consensus on the socket API, a common interface to almost all server programs. It leverages Parrot, a deterministic multithreading system we built, to make multi-threaded replicas deterministic. It uses a new technique we call *time bubbling* to efficiently tackle a difficult challenge of nondeterministic network input timing. Evaluation on five widely used server programs (e.g., APACHE, CLAMAV, and MYSQL) shows that Crane is easy to use, has moderate overhead, and is robust. Crane's source code is at github.com/columbia/crane.

Problem Statement. *State machine replication (SMR)* models a program as a deterministic state machine, where the states are important program data and the transitions are deterministic executions of program code under input requests. SMR runs replicas of the program and invokes a distributed consensus protocol (typically Paxos [59, 57, 90]) to ensure the same sequence of input requests for replicas, as long as a quorum (typically a majority) of the replicas agrees on the input request sequence. Under the deterministic execution assumption, this quorum of replicas must reach the same exact state despite replica failures or network partitions. SMR is proven safe in theory and provides high availability in practice [23, 68, 21, 80, 20, 67, 49, 41].

The fault-tolerant benefit of SMR makes it particularly attractive on implementing a principled replication system for general programs, especially server programs that require high availability. Unfortunately, doing so remains quite challenging; the core difficulty lies in the deterministic state machine abstraction required by SMR, elaborated below.

First, the deterministic execution assumption breaks down in today's server programs because they are almost universally multithreaded. Even on the same exact sequence of input requests, different executions of the same exact multithreaded program may run into different thread interleavings, or *schedules*, depending on such factors as OS scheduling and physical arrival times of requests. Thus, they can easily exercise different schedules and reach divergent execution states – a difficult problem well recognized by the community [17, 49, 45, 41]. To tackle this problem, one prior approach, execute-verify [49], detects divergence of execution states and retries, but it relies on developers to manually annotate states, a strenuous and error-prone process.

Second, to leverage existing SMR systems such as ZooKeeper [4], developers often have to shoehorn their programs into the narrowly defined state machine interfaces provided by these SMR systems. Ideally, experts – those with intimate knowledge of the arcane (think how many papers [59, 57, 90, 23, 68] are needed to explain Paxos), under-specified [68] SMR protocols and subtle failure scenarios in distributed systems – should build a solid SMR system, which all other developers then leverage. However, an SMR system often has to settle for a specific state and transitional interface because it cannot anticipate all possibilities in which developers structure their programs. For example, Chubby [21] defines a lock server interface, and ZooKeeper a pseudo file system interface. Orchestrating a sever program into such a narrow interface not only requires intrusive and error-prone modifications to the program's structure and code, but also disrupts the SMR system itself at times. For instance, developers abused Chubby for storage [21], causing the Chubby developers to add quota support.

Crane. We developed Correctly Replicating Nondeterministic Executions (Crane), an SMR system that transparently replicates server programs for high availability. With Crane, a developer focuses on implementing her program's intended functionality, not replication. When she is ready to replicate her program for availability, she simply runs Crane with her program on multiple replicas. Within each replica, Crane interposes on the socket and the thread synchronization interfaces to keep replicas in sync. Specifically, it considers each incoming socket call (e.g., ACCEPT() a client's connection or RECV() a client's data) an input request, and runs a Paxos consensus protocol [68] to ensure that a quorum of the replicas sees the same exact sequence of the incoming socket calls.

To address nondeterminism, we have built three *deterministic multithreading (DMT)* systems [31, 30, 29]. DMT [18, 33, 73, 15, 17, 45, 16] typically maintains a *logical time* that advances deterministically on each thread's synchronization. Though related, the logical time in DMT is not to be confused with the logical time in distributed systems [58]. By serializing thread synchronizations, DMT practically makes an entire multithreaded execution deterministic. The overhead of DMT is typically moderate because most code is not synchronization and can still run in parallel. Specifically, Crane leverages Parrot [29], our latest DMT system. Parrot incurs merely 12.7% overhead in average on a wide range of 108 popular multithreaded programs on 24-core machines.

A challenge in realizing SMR for multithreaded executions is that, simply combining Paxos and DMT is not sufficient to keep replicas in sync, because the physical time that each request arrives at different replicas may still be different, leading to divergence of execution states and outputs.

Two prior approaches attempted to tackle this challenge. Execute-agree-follow [41] records a partially ordered schedule of Pthreads synchronizations on one replica and replays it on the other replicas, which may incur high network bandwidth consumption and performance overhead. dOS [17] also leverages DMT for replication, but it determines the logical admission time for each request using two-phase commit. Aside from two-phase commit's known intolerance of primary failures, per-request commit is also costly.

One may consider solving this challenge by leveraging the underlying distributed consensus protocol to determine the logical admission time for each request. Specifically, when running the consensus protocol to decide each request's position in the request sequence, a predicted logical admission time can be carried as part of the decision as well. Unfortunately, predicting a logical admission time for each request accurately is quite challenging because typical server programs have background threads which may frequently tick logical clocks. A too-small predication leads to replica divergence if another replica has already run past the predicted logical time. A too- large predication blocks the system unnecessarily because replicas cannot admit the request before reaching the predicted time.

Our key insight is that many requests need no admission time consensus because their admission times are already deterministic. Hypothetically, if the requests arrive faster than they are admitted at each replica, each request's admission time is fully deterministic because each replica simply admits requests as fast as it can. In practice, requests do not arrive this fast. However, there are still frequent bursts of requests that arrive together. Among replicas, as long as the first request of a burst is admitted at a deterministic logical time, all the other requests in the burst are admitted at deterministic logical times without requiring consensus.

Leveraging this insight, we created a technique called *time bubbling* to enforce deterministic logical times efficiently. It ensures that the first request in a burst is admitted at each replica deterministically by inserting a deterministic wait after the previous burst of requests are all admitted. During this wait, each replica only processes already admitted requests, and does not admit new requests. Crane negotiates a consistent duration of the wait via the underlying distributed consensus protocol, and enforces this wait at each replica via DMT. These waits are like deterministic time bubbles between bursts of requests (hence the name of the technique), creating the illusion that the requests arrive faster than they are admitted.

In short, by converting per-request admission time consensus to per-burst, time bubbling efficiently combines the input determinism of Paxos and the execution determinism of DMT. For busy servers, requests in bursts greatly outnumber the other requests. (We observed that 66.65% to 93.88% of requests are in bursts.) They rarely need to invoke time consensus, enjoying good performance. For idle servers, time consensus overhead does not matter much because the servers are idle anyway.

Evaluation. We implemented Crane by interposing on the POSIX socket and the Pthreads synchronization interfaces. It intercepts operations along these interfaces by hijacking dynamically linked library calls for transparency. It implements the Paxos protocol atop libevent [60] for distributed consensus, and leverages our Parrot system for deterministic multithreading. Unlike prior SMR systems with narrow interfaces, Crane's checkpoint and recovery must work with general programs. To this end, it leverages CRIU [26] to checkpoint and restores process states and LXC [2] for file system states. An additional benefit of using the LXC container is that Crane isolates the replicated server program from the environment, avoiding nondeterministic resource contentions.

We evaluated Crane on five widely used server programs, including HTTP servers APACHE and MONGOOSE, an anti-virus server CLAMAV, an uPnP multimedia server MEDIATOMB, and a database server MYSQL. Our results on popular performance benchmarks show that Crane works with all the servers easily (three servers require no modification, and the other two servers each require only two lines of Parrot hints [29] to improve performance); that Crane's performance overhead is moderate (an average of 34.19% overhead at the servers' peak performance setups on our 24-core machines); and that Crane is robust on replica failures.

Our evaluation covered a number of questions, including: (1) Is Crane easy to use? (2) Compared to nondeterministic executions, does Crane consistently enforce the same sequence of network outputs among replicas? (3) What is Crane's performance overhead compared to nondeterministic executions? (4) When the default schedules enforced by the Parrot DMT scheduler are slow, how much optimization can Parrot's performance hints bring to Crane? (5) How sensitive are the two time bubbling parameters to Crane's performance? (6) How fast are Crane's checkpoint and recovery components on handling replica failures? For the purpose of this report, we will only discuss question (3); we refer the reader to our Crane paper [27] for more detailed evaluation.

To understand the performance impact of Crane's components, we divided Crane's components into two major parts: the DMT part ran by Parrot; and the proxy (with Paxos) part which enforces the same sequence of client socket calls across replicas. Each part ran independently without the other part. The proxy part represents the performance overhead of invoking Paxos consensus for client socket calls, and the DMT part represents the Parrot DMT scheduler's overhead.

Figure 14 shows the servers' performance running in Crane normalized by their unreplicated nondeterministic executions. The mean overhead of Crane for the five evaluated programs is 34.19% due to two main reasons. First, except for MYSQL, which does fine-grained, per-table mutex and read-write locks frequently, the DMT schedules were efficient on the other four servers. The reason is that Parrot's scheduling primitives are already highly optimized for multi-core [29].

The proxy-only part incurred 0.82%~3.46% overhead, which is not surprising, because the number of socket calls is much smaller than the number of Pthreads synchronizations in these programs. In short, Crane's performance mainly depends on the DMT schedules' performance.

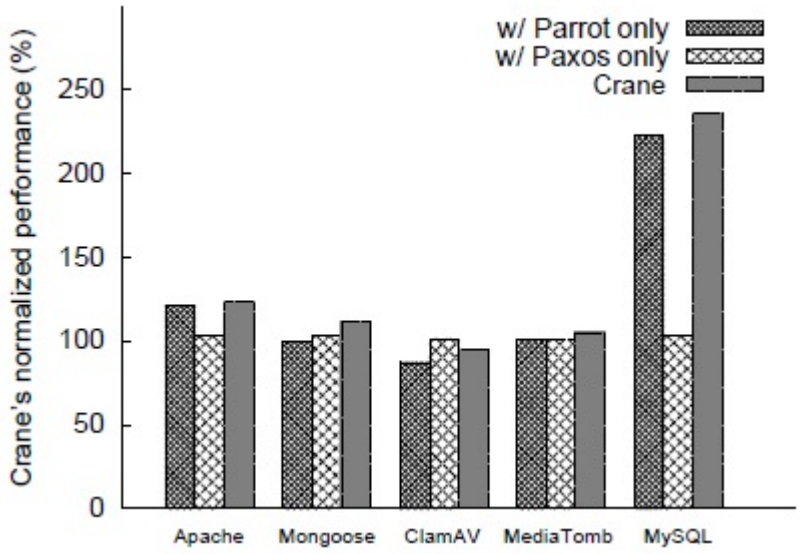


Figure 14: Crane Performance - Times are normalized to unreplicated, nondeterministic execution

MEDIATOMB incurred modest speedup because its transcoder MENCODER had significant speedup with Parrot. We inspected MEDIATOMB's micro performance counters with the Intel VTUNE [92] profiling tool. When running in Crane, MEDIATOMB only made 6.6K synchronization context switches, while in the Pthreads runtime it made 0.9M synchronization context switches. This saving caused MEDIATOMB running with Parrot a 12.76% speedup compared to its nondeterministic execution. The Parrot evaluation [29] also observed a 49% speedup on MENCODER.

The time bubbling technique saves most of needs on invoking consensus for the logical times of clients' socket operations, confirmed by the low frequency of inserted time bubbles in Table 2. APACHE, MEDIATOMB, and MONGOOSE uses APACHEBENCH as its benchmark, and each request contained a CONNECT(), SEND(), and CLOSE() call. CLAMAV uses its own CLAMDSCAN bench- mark, and each request contained 18 socket calls. MYSQL's benchmark contained 6~7 socket calls for each query. The ratio of inserted bubbles is merely 6.12%~33.35%. MEDIATOMB had the highest ratio of time bubbles because it took the longest time (9,703ms) to process each request.

Table 2: Time bubbles

Program	# client socket calls	# time bubbles	%
APACHE	3,000	450	13.04
CLAMAV	18,000	1,173	6.12
MEDIATOMB	3,000	1,501	33.35
MONGOOSE	3,000	448	12.99
MYSQL	6,750	573	7.82

4.6 Hardware-Enhanced Memoization Technologies

The MEERKATS architecture relies upon high levels of replication to improve cloud resilience. Some of the replicas maintained in by a MEERKATS clouds are identical copies of the data (as maintained with Crane, see Section 4.5). Other replicas are fake, decoy replicas of services, requests, and data (as created and maintained with DIGIT, Novo, and SAuth, see Section 4.3).
A key challenge in maintaining a high degree of replication – whether real or fake – is to deal with significant added redundancy efficiently. If we have to run instead of one copy of a service, five or ten copies of it, how can we do so efficiently?

As part of the MEERKATS project we investigated using compiler and hardware optimizations to make it efficient to run highly redundant computations. Specifically, we developed *CHAMP*, a hardware/software framework for complexity-effective program memoization. The tool consists of a compiler transformation to modify select functions to record their inputs and supply outputs immediately if previously observed invocation is recognized. When past executions are invoked again (e.g., to create uncertainty), the system returns their results without expending the resources to re-compute. Of course, it is not only possible to reuse prior executions, and our work establishes the conditions when reuse can happen versus not.

Our main goal in CHAMP was to develop infrastructure for *automatically memoizing* the program execution. The project proceeded in two phases. The first explored value locality and reuse in unmodified programs. The second phase created a *compiler transformation* based on the Clang/LLVM compiler suite to automatically insert memoization instructions into unmodified programs. Given poor performance of a pure software approach, the prototype was extended to contain a *supplementary instruction set* to perform memoization operations efficiently. The transformation seeks to exploit functions whose effects on machine state can be entirely captured by its loads and stores. Under CHAMP, such functions are called *reproducible* and can be deterministically replayed by setting register contents and issuing memory writes. Program performance improves when accessing and replaying the function's side-effects is cheaper than recalculating them.

4.6.1 Value Reuse Measurement

Before attempting to automatically memoize programs, it was necessary to determine that value reuse existed in programs. Without sufficient value locality, memoization would be ineffective.

It is important to differentiate value *reuse* and value *locality*. Value reuse measures the recurrence of particular values for a *function object* across separate invocations of that function. A function object is any value obtained by the function that it does not generate itself. This means global variables, function arguments, and any values loaded through a pointer. It is important to note that only static loads are considered; each dynamic load does not create a new object to be tracked. It is helpful to think of how a compiler analyzes and predicts the value evolution of a loop counter; each separate loop execution still corresponds to the same variable, even though the value has likely changed.

Value reuse tracks the variance of in value of each function object. Value locality measures how often a value reappears across subsequent executions, analogous to temporal locality for the purposes of general-purpose memory caching. Value reuse, combined with the total number of distinct values observed, gives a rough estimate of the storage upper-bound required to store all repeated values, while locality provides intuition on whether or not the reuse can be effectively exploited with a real cache.

Measurement Tool. Two tools with similar purposes were created to measure value locality. The first is a Pintool to identify the function objects in the executing program and to dump a trace containing their values. A small transformation in LLVM was also implemented to accomplish the same task, in the case where the overhead of Pin is unacceptable – or to handle concurrent programs (the version of Pin at time of implementation had subtle deadlock bugs when used concurrently). Both tools produce traces in identical format. These traces are then consumed by a tool, '*memo-dump*', which creates a graphical representation of the value samples, as well as a textual summary of their variance and other statistical properties.

Measurement Results. Several programs were executed with these tools. The resulting traces were analyzed by *memo-dump* and trends observed. The program objects generally fell into three categories: (1) *near-random*, with little or no value reuse; (2) *near-constant*, where the value remained the same over program execution or changed very rarely, immediately returning to the previously observed value; and (3) *cycling*, the object's value followed a short cycle of two to four values.

Apache Web Server Results: Running the Apache webserver was aimed to prove several points, chiefly that the tool could analyze a complex, real-world application. Input data was also supplied to check that value reuse varied appropriately. For example, when fetching the same static page repeatedly, there should be high value reuse, as the same request from the client is parsed and the same page delivered. Conversely, random HTML and random PNG images were created by a script and then fetched, resulting in a very low level of value reuse. A strong indicator of accuracy is in both cases, Apache's own internal caching mechanism showed very high value reuse.

SPEC CPU2006 Results: The SPEC CPU2006 integer suite was monitored with the tool and showed results that varied largely across benchmarks. Benchmarks showing high reuse include h264ref and bzip2. These both perform compression and decompression operations, which seek to take advantage of low-entropy, repetitive sequences in the input stream. The 'dictionary' style representation – substituting a shorter sequence for a more complicated one – gives a high level of value reuse in accessing and outputting these values.

Other Benchmarks: Other benchmarks exhibited low reuse, such as perlbench and gcc. These benchmarks performed seemingly repetitive tasks of parsing, but use many global variables and other spaghetti code structuring. This possibly biases the results by creating separate function objects for what is truly one object in the source code. In any case, this would likely not be exploitable by a memoization scheme without very precise alias analysis to disambiguate the objects.

Take-Away: These tools gave strong confidence that memoization was feasible and could be made profitable. The next chapters describe the design and implementation of the compiler analysis and transformation, as well as the decision to accelerate the framework with hardware extensions.

4.6.2 Compiler-Enhanced Memoization

LLVM was augmented with several analysis and transformation passes which take unmodified LLVM intermediate representation (IR) and add the necessary instrumentation for memoization. Since the passes operate on IR, the compiler can instrument any programs in a language translatable to LLVM IR. Additionally, the instrumentation at the IR level makes no assumption about the architecture or execution environment, and so could be accelerated by any architecture which LLVM targets.

Compiler Design. The analysis pass works to split the single-entry single-exit (SESE) basic blocks of a function's control flow graph into an input set and output set. The input set contains all blocks that perform input operations, or which flow to such blocks. The output set contains the complement of the input graph – any blocks whose edges cannot flow to input operations.

On function entry at runtime, the input set is mapped to the output set by a *memoization store*, a simple cache for each function. If inputs were seen on a previous invocation, the outputs are applied directly from the memoization store and the instructions that calculate the outputs are elided.

Reproducibility is determined by the properties of the function's individual instructions. If all of a function's instructions are reproducible, then the function itself is reproducible. Call instructions are a special case. Intuitively, their reproducibility is determined by the function they call. Thus, a reproducible function may only call other reproducible functions.

If a function's inputs are trivially calculated the function's invocation may also be elided entirely by a calling function. More precisely, if the function is to be elided in the caller, the call instruction occurs after the caller's input collection is complete and the memoization store is accessed. To be considered trivial, all of the caller's input must be safe to reorder before this access. Calls to such functions that satisfy these conditions are *elidible* and the functions themselves are *call-elidible*.

The two instruction streams which calculate the input and output sets, respectively, may not overlap and must be separated by the access to the memoization store. The input instruction stream will capture all loads and non-elidible calls in the function. The output instruction stream is simply the remainder of the function. Note that stores may occur in the input instruction stream, but they are not considered output. This is correct, as they will be performed before the access to the memoization store, so they do not need to be recorded. This restriction prevents the memoization of systems that perform user space threading (also known as green threads or fibers), as these would interleave access to memoization threads. This tradeoff is necessary as a mechanism to associate instructions with their respective caches was considered too costly.

Very few functions in the average program will be reproducible according to this strict definition. Notably, error handling and logging will invalidate a function's execution by writing to the terminal or a log file. However, errors intuitively will occur only in exceptional cases. Therefore, by introducing a mechanism to dynamically invalidate function executions, code coverage can be increased while preserving original program semantics.

To only invalidate control flow paths that contain invalid instructions, basic blocks are marked invalid if and only if they contain them. The invalidation instrumentation may not reside directly in the invalid blocks for efficiency reasons. For example, a loop that contains an unconditionally executed invalid block will have the invalidation at the loop's entry, rather than in the loop body, to avoid re-execution. When invalidation instrumentation is executed, any state being gathered for the execution is flushed and no modification to the memoization store is made. When a function is invalidated, all functions in the call stack that are also instrumented will have their current execution invalidated. That is, invalidating a function execution also invalidates its callers' executions.

Evaluation. We evaluated the compiler optimizations on all of the SPEC2006 benchmarks on real hardware. While we observed good potential for value reuse in several benchmarks, the overheads were impractically high (1.4x - ~50x) mainly due to the overhead of maintaining the memoization caches. Based on these results we concluded hardware support was necessary.

4.6.3 Hardware-Enhanced Memoization

To further improve performance, we designed a set of new primitive operations that can be implemented in hardware.

Hardware Design. Memoization support is composed of several primitive operations:

(1) Region entry and exit: hardware instructions signal when a memoized region is entered or exited. Entrance is signaled explicitly, while exit is implicit at either a hit in, or commit to, the memoization store.

(2) Input hashing: Registers and memory locations are appended to the input set to be hashed by a special functional unit.

(3) Lookup: The accumulated hash is used to index into the memoization store. The instruction will set a flag in a destination register indicating whether there was a hit in the memoization store.

(4) Record side-effects: Adds registers or memory stores to a hardware buffer to be committed to the memoization store.

(5) Commit: Associates recorded side-effects with the accumulated hash in the memoization store.

(6) Apply: After a hit in the memoization store, applies the recorded side-effects to the machine state.

In addition to the memoization store and the aforementioned hash unit, there are two parallel hardware buffers that operate as stacks. The first buffer stores side-effects which have been recorded by functions, but have not yet been committed to the memoization store. Stores that occur as part of an output instruction stream, as well as explicitly indicated registers, will be appended here. The second buffer stores control data concerning the first buffer. For each memoized function pushed onto the stack, a fixed size entry is created that records a byte offset into the buffer, which along with the top of the buffer, delimits the side-effects that belong to that function. The capacity of this control stack determines the maximum nesting depth for the architecture. When a new function is pushed

and no new entries are available, the oldest entry will be overwritten. As the depth counter saturates at the maximum depth, this silently discards the oldest function, but the younger functions are unaffected. When the first buffer overflows, the processor will respond as if an invalid memoization operation has occurred. The effects of this are described in the next section.

Operation. The memoization hardware contains a control register that determines whether any memoization regions are active (ENBL bit), whether an input or output stream is being executed (RW bit; cleared for read), and a saturating memoization region nesting depth counter. When a memoized function is entered, a CHMPENTER instruction is executed. ENBL is set and RW is cleared. The depth counter is also incremented, saturating at a microarchitectural specific maximum depth. If the counter saturates, the "oldest" memoization frame is silently discarded. The implications of this will become clear when discussing the commit operation.

This begins the function's input instruction stream. Argument values contained in registers will be hashed with CHMPHASHREG. Loaded values are automatically hashed while the RW bit is cleared and ENBL is set. Since a hash operation by design cannot be rolled back, it is necessary for these instructions to have in-order non-speculative execution.

At the end of input operations, control will flow to a CHMPLKUP operation, which looks up with the accumulated hash into the memoization store. This will operation will of course depend on the hash being computed, a potentially lengthy operation. The RW bit is set, as the input stream is completed. If the memoization store contains an entry matching the hash, a record to the entry is stored for application by a CHMPAPPLY instruction. This instruction also retrieves the return value.

If there is a miss in the memoization store, control will flow to the output stream. Similarly to loads, stored values will automatically be recorded when the RW and ENBL bits are set. Additionally, registers that are used for return values can be recorded with CHMPSTREG.

Finally, a CHMPCOMMIT instruction is reached immediately before function return. This commits all recorded data to the memoization store with the accumulated hash. For both the CHMPAPPLY and CHMPCOMMIT instructions, the memoization depth counter is decremented. If the counter is zero, then the ENBL bit is cleared.

When an invalidation instruction, CHMPINVL, is executed, the ENBL bit is cleared and the memoization depth is set to zero, reflecting that invalidation affects all scopes. The hardware buffer entries that store memoization data do not need individual valid bits since they are indexed using the memoization depth. Until a CHMPENTER is executed, the cleared ENBL bit forces all other memoization operations to be nops. When the ENBL bit is again set, the memoization depth will be at zero, so the buffers can be indexed normally.

Preliminary Evaluation. The memoization hardware extension is implemented inside the gem5 cycle-accurate microarchitecture simulator. The instructions were added as an extension to the x86-64 instruction set (sometimes known as AMD64 or x64). The LLVM x86-64 backend was modified to emit instruction inside the "hinting NOP" opcode space. These opcodes are reserved by Intel and

have been used to accommodate the cache prefetch opcodes as well as the Transactional Synchronization eXtensions (TSX). Older processors or models which do not support these extensions simply decode these instructions as NOPs. A carefully transformed program can be run on a processor without the memoization extensions and would retain its original semantics.

Preliminary evaluation on microbenchmarks indicate that the proposed hardware optimizations can in fact mitigate many of the software induced performance overheads. Future evaluation will include full benchmarks on platforms that support fast simulation speed.

4.7 New Attacks

As a final class of contributions as part of the MEERKATS program, we discovered a set of new kinds of attacks. These attacks were not included in our initial proposal, but we present them as new significant findings within the scope of our initial vision.

4.7.1 ARC: Protecting against HTTP Parameter Pollution Attacks

HTTP Parameter Pollution Attacks. HTTP Parameter Pollution (HPP) is a recently discovered technique for exploiting web applications. HPP can be considered as an injection attack that targets URLs; one of the fundamental concepts of the web platform. Web browsers communicate with web applications through HTTP requests and responses, which reference resources using URLs. This communication can be polluted by injecting parameters in the HTTP stream. These injected parameters form URLS, which if served, instruct the application to perform actions that were originally not part of the application's design. Thus, the control flow of the web application is altered according to an attacker's need.

To illustrate the attack in a short example, consider an e-store application taking two arguments, namely: a product identifier and an action, which affects the product state. A combination of a product identifier and the action purchase results in ordering a product. The product identifier and the action must be attached as parameters in a URL, which in turn is communicated to the application through the construction of an HTTP request. If the attacker manages to pollute the request with extra parameters, then the control flow of the application may change in numerous ways. The simplest manifestation of the vulnerability is for the attacker to inject a particular parameter multiple times. In case the parameter that carries out the product identifier is duplicated, then many different control flows can take place, depending on the parameter occurrence (first, last, or a combination of) that the application will give significance while the URL is parsed.

About 1,499 of 5,000 highly ranked in Alexa.com web sites are considered vulnerable to HPP exploitation according to the methodology outlined by Balduzzi et al.[14].

The ARC Defense. We have developed Application Request Cache (ARC), a framework that can protect web applications from HPP exploitation. ARC does not detect HPP vulnerabilities, although it can record HPP exploitation attempts. It is deployed at server side and works in a transparent way. In essence, a web application can be protected, from HPP exploitation by simply incorporating ARC in

the application server. Clients need no further modifications. In contrast to PAPAS [14], which currently is the only available methodology for discovering HPP vulnerabilities, ARC aims at protecting the web application without auditing. ARC assumes that the web application is vulnerable and tries to protect it from being exploited. To this respect, ARC and PAPAS can be combined. The former as a protection layer and the latter as a periodic auditor.

ARC is based on the following fundamental concept. Each web application is characterized by a set of URL schemas, which act as generators of the complete functional set of URLs that compose the application's logic. A URL schema is extracted by a URL by masking out all variables that are assigned to the URL's parameters. Each control flow is triggered by having the application serving a URL, which stems from a particular URL schema. ARC collects all schemas taken from benign requests during a training phase. At production time, for each incoming request, ARC extracts the URL and its schema, and searches for it in a set of already known benign schemas. In case the schema is not found, the request is rejected and the event is recorded. An incoming polluted URL will have no schema stored in ARC and thus will be rejected. This methodology cannot only prevent HPP, but also certain types of XSS, where JavaScript is attached to HTTP parameters.

Evaluation. Our ARC prototype was developed using Google's Go, a very efficient programming language for constructing system tools. In our implementation, ARC stores all cached schemas in carefully selected data structures, which are implemented using maps and slices, as provided by Go. ARC also takes advantage of the multiprocessing features of Go, goroutines and channels. In a 4-core Linux server, ARC can process hundreds of thousands of URLs per second. A typical request resolution takes no more than a few microseconds.

4.7.2 CellFlood: A New Attack Against Tor Onion Routers

Context. To date, the Tor network, one of the most widely used anonymizing systems, consists of more than 3000 Onion Routers that serve daily over 400000 users. Being a distributed system operated by volunteers, the anonymity of Tor users is vulnerable to attacks where a set of malicious routers, controlled by an adversary, join the network with the aim of gaining control of user circuits. The Tor network is specifically designed and continuously updated to address these types of threats, but another option available to the adversary would be that of putting a network DoS attack into place with the aim of making it impossible, or very hard, for users to communicate with Tor routers and Tor routers with each other. Such an attack could be used to either significantly degrade the users' perceived quality of service, which would discourage them from using Tor, or to affect the topology of the Tor network in a way that favors traffic flowing through malicious routers, thus increasing the power of the adversary. A network DoS would not require a deep knowledge of the Tor network internals and could be performed by using well known, pre-existing, off-the-shelf methods. Clearly, since such an attack is orthogonal to those that the Tor network was designed to address, we should not expect Tor to be resilient to it.
Nevertheless, the protocols used by clients to setup circuits through the Tor network are vulnerable to a simple attack that would allow an adversary to achieve an effect similar to that of a network DoS, but with just a fraction of its bandwidth resources.

CellFlood. We have designed an attack, named *CellFlood*, and provide an experimental evaluation

of it both in a controlled environment and on the real Tor network. Our results, and our estimations based on measurements from a real Tor router, showed that CellFlood is not only effective, but also cheap enough to make a feasible alternative to more sophisticated attacks to the Tor network that have been presented in the past. As a way to mitigate the effect of this attack, we proposed to use a client puzzle-based technique that would allow Tor routers under attack to keep their ability to provide service to honest clients. Our improved version of the protocol allows routers under attack to easily impose a cap on the attacking host(s), thus preserving their ability to process honest client requests. At the same time, our tests confirm that our protocol has a small impact on the quality of the service perceived by Tor users, even in extreme scenarios

Evaluation. We have conducted an extensive evaluation regarding the feasibility of the attack both in a controlled environment as well as on the real Tor network. Our findings demonstrated that the attack is effective under different configuration parameters.

Also, we have introduced a lightweight estimation technique for the resilience of a remote, non-cooperative Onion Router to the attack. Our estimates showed that, to halve the processing capability of 62% of the most used Onion Routers of the Tor network, CellFlood would require between 2.6 and 9.76 Mb/s per router.

5 CONCLUSIONS

In the course of the MEERKATS project, we have designed, implemented, evaluated, and in some cases deployed a set of new technologies that enable continuous change, deception, and unpredictability of cloud environments as a way of increasing their resiliency to a wide range of attacks.

These technologies present significant advances along five major directions:

1. *Continuous migration technologies* that can enable for the first time the swift migration of cloud-resident services and data either in response to an attack or continuously so as to present a moving-target defense;
2. *Cloud information flow tracking technologies* that can track cloud-resident data at larger scales than ever before, enabling cloud users (e.g., service administrators) to audit the flow of their information in the cloud;
3. *Misinformation and decoy technologies* that can automatically generate deceptive information – bogus information that appears genuine – so as to confuse, bait, and track attackers;
4. *Cloud monitoring and self-healing technologies* that can integrate information from many sensors spread across the cloud to detect complex, multi-stage attacks; and
5. *Stable multithreading technologies* that can reduce the security risks posed by concurrent programs by ensuring that upon every execution, a program takes one of a few pre-checked schedules that have already been validated as safe.
6. *Hardware-enhanced memoization technologies* that can reduce the performance overheads of running enormous additional redundant or fake computations by enabling safe reuse of previously derived values.

We have already started a transition of some of these technologies (notably the decoy technologies) as part of this project. While more work remains to be done to roll out all of these technologies into a production cloud environment; we believe that in the future, they will become the bases for crucial components of future resilient clouds.

6 PUBLISHED PAPERS

1. Urgent virtual machine eviction with enlightened post-copy. Yoshihisa Abe, Roxana Geambasu, Kaustubh Joshi, and Mahadev Satyanarayanan. Proceedings of the Virtual Execution Environments Conference (VEE), Atlanta, GA, April 2016.

2. Towards the Detection of Multistage Attacks Using Cross-Layer Alerts Correlation. Hang Zhao, Adrian Tang, Nathaniel Boggs, Salvatore J. Stolfo. Under submission. Draft available upon request.

3. Paxos Made Transparent. Heming Cui, Rui Gu, Cheng Liu, Tianyu Chen, Junfeng Yang. Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15), October, 2015.

4. Secure Deduplication of General Computations. Yang Tang, Junfeng Yang. Proceedings of the USENIX Annual Technical Conference (USENIX ATC '15), 2015.

5. Sunlight: Fine-grained Targeting Detection at Scale with Statistical Confidence. Mathias Lecuyer, Riley B. Spahn, Giannis Spiliopoulos, Augustin Chaintreau, Roxana Geambasu, and Daniel Hsu. In Proceedings of the ACM Conference on Computer and Communications Security (CCS), Denver, Colorado, October 2015.

6. Model Aggregation for Distributed Content Anomaly Detection. Sean Whalen, Nathaniel Boggs, Salvatore J. Stolfo. In the Proceedings of the 7th ACM Workshop on Artificial Intelligence and Security (AISec), November 2014.

7. Pebbles: Fine-Grained Data Management Abstractions for Modern Operating Systems. Riley Spahn, Jonathan Bell, Michael Lee, Sravan Bhamidipati, Roxana Geambasu, Gail Kaiser. In the Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI), October 2014.

8. Unsupervised Anomaly-based Malware Detection using Hardware Features. Adrian Tang, Simha Sethumadhavan, Salvatore Stolfo. In the Proceedings of the 17th International Symposium on Research in Attacks, Intrusions and Defenses (RAID), September 2014.

9. Synthetic Data Generation and Defense in Depth Measurement of Web Applications. Nathaniel Boggs, Hang Zhao, Senyao Du, Salvatore J. Stolfo In the Proceedings of the 17th Inter- national Symposium on Research in Attacks, Intrusions and Defenses (RAID), September 2014.

10. XRay: Increasing the Web's Transparency with Differential Correlation. Mathias Lecuyer, Guillaume Ducoffe, Francis Lan, Andrei Papancea, Theofilos Petsios, Riley Spahn, Augustin Chaintreau, and Roxana Geambasu. In the Proceedings of the 23rd USENIX Security Symposium, August 2014.

11. Catch Me if You Can: A Cloud-Enabled DDoS Defense. Quan Jia, Huangxin Wang, Dan Fleck, Fei Li, Angelos Stavrou, Walter A. Powell. In the Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (IEEE DSN 2014), Atlanta, Georgia USA, June 23 - 26, 2014.

12. A Moving Target DDoS Defense Mechanism. Huangxin Wang, Quan Jia, Dan Fleck, Walter Powell, Fei Li, Angelos Stavrou. In the Elsevier Journal of Computer Communications, Volume 46, 15 June 2014, Pages 10-21, ISSN 0140-3664(2014).

13. SAuth: Protecting User Accounts from Password Database Leaks George Kontaxis, Elias Athanasopoulos, Georgios Portokalidis, and Angelos D. Keromytis. In the Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS), November 2013.

14. Parrot: a Practical Runtime for Deterministic, Stable, and Reliable Threads Heming Cui, Jiri Simsa, Yi-Hong Lin, Hao Li, Ben Blum, Junfeng Yang, Garth A. Gibson, and Randal E. Bryant. In the Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP), November 2013.

15. vTube: Efficient Streaming of Virtual Appliances Over Last-Mile Networks. Yoshihisa Abe, Roxana Geambasu, Kaustubh Joshi, H. Andres Lagar-Cavilla, and Mahadev Satyanarayanan. In the Proceedings of the ACM Symposium on Cloud Computing (SOCC), October 2013.

16. CloudFence: Data Flow Tracking as a Cloud Service. Vasilis Pappas, Vasileios P. Kemerlis, Angeliki Zavou, Michalis Polychronakis, and Angelos D. Keromytis. In the Proceedings of the 16th International Symposium on Research in Attacks, Intrusions and Defenses (RAID), October 2013.

17. CellFlood: Attacking Tor Onion Routers on the Cheap. Marco Valerio Barbera, Vasileios P. Kemerlis, Vasilis Pappas, and Angelos D. Keromytis. In the Proceedings of the 18th European Symposium on Research in Computer Security (ESORICS), September 2013.

18. Determinism Is Not Enough: Making Parallel Programs Reliable with Stable Multithreading. Junfeng Yang, Heming Cui, Jingyue Wu, Yang Tang, Gang Hu. In Communications of the ACM, 2013.

19. Effective Dynamic Detection of Alias Analysis Errors. Jingyue Wu, Gang Hu, Yang Tang, Junfeng Yang. In the Proceedings of the 9th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC-FSE), August 2013.

20. Cloud resiliency and security via diversified replica execution and monitoring. Azzedine Benameur, Nathan S. Evans, and Matthew C. Elder. In the Proceedings of the 6th International Symposium on Resilient Control Systems (ISRCS), August 2013.

21. MOTAG: Moving Target Defense Against Internet Denial of Service Attacks. Quan Jia, Kun Sun, and Angelos Stavou. In the Proceedings of the 22nd International Conference on Computer Communications and Networks (ICCCN), July 2013.

22. Cloudopsy: an Autopsy of Data Flows in the Cloud. Angeliki Zavou, Vasilis Pappas, Vasileios P. Kemerlis, Michalis Polychronakis, Georgios Portokalidis, and Angelos D. Keromytis. In the Proceedings of the Human Computer Interaction International 2013 (HCI), July 2013.

23. On the Feasibility of Online Malware Detection with Performance Counters. John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Simha Sethumadhavan, and Sal Stolfo. In the

Proceedings of the 40th International Symposium on Computer Architecture (ISCA), June 2013.

24. Sound and Precise Analysis of Parallel Programs through Schedule Specialization. Jingyue Wu, Yang Tang, Gang Hu, Heming Cui, Junfeng Yang. In the Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), June 2012.

25. Concurrency Attacks. Junfeng Yang, Ang Cui, Sal Stolfo, and Simha Sethumadhavan. In the Proceedings of the 4th USENIX Workshop on Hot Topics in Parallelism (HOTPAR), June 2012.

26. CleanOS: Limiting Mobile Data Exposure with Idle Eviction. Yang Tang, Phillip Ames, Sravan Bhamidipati, Ashish Bijlani, Roxana Geambasu, and Nikhil Sarda. In the Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI), October 2012.

27. Self-healing Multitier Architectures Using Cascading Rescue Points. Angeliki Zavou, Georgios Portokalidis, and Angelos D. Keromytis. In the Proceedings of the 2012 Annual Computer Security Applications Conference (ACSAC), December 2012.

28. Adaptive Defenses for Commodity Software through Virtual Application Partitioning. Dimitris Geneiatakis, Georgios Portokalidis, Vasileios P. Kemerlis, and Angelos D. Keromytis. In the Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS), October 2012.

29. Lost in Translation: Improving Decoy Documents via Automated Translation. Jonathan Voris, Nathaniel Boggs, and Salvatore J. Stolfo. In the Proceedings of the Workshop on Research for Insider Threat (WRIT), May 2012.

30. Fog Computing: Mitigating Insider Data Theft Attacks in the Cloud. Salvatore J. Stolfo, Malek Ben Salem, and Angelos D. Keromytis. In the Proceedings of the Workshop on Research for Insider Threat (WRIT), May 2012.

31. Towards a Universal Data Provenance Framework using Dynamic Instrumentation. Eleni Gessiou, Vasilis Pappas, Elias Athanasopoulos, Angelos D. Keromytis, and Sotiris Ioannidis. In the Proceedings of the 27th IFIP International Information Security and Privacy Conference (SEC), June 2012.

32. Position Paper: The MEERKATS Cloud Security Architecture. Angelos D. Keromytis, Roxana Geambasu, Simha Sethumadhavan, Salvatore J. Stolfo, Junfeng Yang, Azzedine Benameur, Marc Dacier, Matthew Elder, Darrell Kienzle, and Angelos Stavrou. In the Proceedings of the 3rd International Workshop on Security and Privacy in Cloud Computing (ICDCS-SPCC), June 2012.

REFERENCES

[1] Amazon web services. http://aws.amazon.com.

[2] LXC. https://linuxcontainers.org/.

[3] VMware vSphere vMotion Architecture, Performance and Best Practices in VMware vSphere 5. http://www.vmware.com/files/pdf/vmotion-perf- vsphere5.pdf.

[4] ZooKeeper. https://zookeeper.apache.org/.

[5] Y. Abe, R. Geambasu, K. Joshi, and M. Satyanarayanan. Urgent virtual machine eviction with enlightened post-copy. In *Proceedings of the Virtual Execution Environments Conference (VEE)*, Atlanta, GA, April 2016.

[6] B. Adida. Beamauth: Two-factor web authentication with a bookmark. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 48–57, New York, NY, USA, 2007. ACM.

[7] P. Akritidis. Cling: A memory allocator to mitigate dangling pointers. In *Proceedings of the 19th USENIX Conference on Security*, USENIX Security'10, pages 12–12, Berkeley, CA, USA, 2010. USENIX Association.

[8] D. G. Andersen. Mayday: distributed filtering for internet services. In *USITS'03: Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems*, pages 3–3, Berkeley, CA, USA, 2003. USENIX Association.

[9] R. Anderson and M. Kuhn. Tamper resistance: A cautionary note. In *Proc. of the USENIX Workshop on Electronics Commerce*, 1996.

[10] T. Anderson, T. Roscoe, and D. Wetherall. Preventing internet denial-of-service with capabilities. *SIGCOMM Comput. Commun. Rev.*, 34(1):39–44, 2004.

[11] Apple iCloud. Find my iPhone, iPad, and Mac. www.apple.com/icloud/features/find- my-iphone.html, 2012.

[12] T. Aura, P. Nikander, and J. Leiwo. Dos-resistant authentication with client puzzles. In *Security Protocols Workshop*, pages 170–177, 2000.

[13] M. Baker-Harvey. Google compute engine uses live migration technology to service infrastructure without application downtime.

[14] M. Balduzzi, C. T. Gimenez, D. Balzarotti, and E. Kirda. Automated discovery of parameter pollution vulnerabilities in web applications. In *NDSS*. The Internet Society, 2011.

[15] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: a compiler and runtime system for deterministic multithreaded execution. In *Fifteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '10)*, pages 53–64, Mar. 2010.

[16] T. Bergan, L. Ceze, and D. Grossman. Input-covering schedules for multithreaded programs. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, pages 677–692. ACM, 2013.

[17] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble. Deterministic process groups in dOS. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, Oct. 2010.

[18] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel java. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '09)*, pages 97–116, Oct. 2009.

[19] N. Boggs, H. Zhao, S. Du, and S. J. Stolfo. Synthetic data generation and defense in depth measurement of web applications. In *17th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2014.

[20] W. J. Bolosky, D. Bradshaw, R. B. Haagens, N. P. Kusters, and P. Li. Paxos replicated state machines as the basis of a high-performance data store. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, Berkeley, CA, USA, 2011. USENIX Association.

[21] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the Seventh Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 335–350, 2006.

[22] M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 147–160, Berkeley, CA, USA, 2006. USENIX Association.

[23] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: An engineering perspective. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing (PODC '07)*, Aug. 2007.

[24] J. Cheney, L. Chiticariu, and W.-C. Tan. Provenance in databases: Why, how, and where. *Found. Trends databases*, 1(4):379–474, Apr. 2009.

[25] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2005.

[26] Criu. http://criu.org, 2015.

[27] H. Cui, R. Gu, C. Liu, and J. Yang. Paxos made transparent. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*, Oct. 2015.

[28] H. Cui, G. Hu, J. Wu, and J. Yang. Verifying systems rules using rule-directed symbolic execution. In *Eighteenth International Conference on Architecture Support for Programming*

Languages and Operating Systems (ASPLOS '13), 2013.

[29] H. Cui, J. Simsa, Y.-H. Lin, H. Li, B. Blum, X. Xu, J. Yang, G. A. Gibson, and R. E. Bryant. Parrot: a practical runtime for deterministic, stable, and reliable threads. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Nov. 2013.

[30] H. Cui, J. Wu, J. Gallagher, H. Guo, and J. Yang. Efficient deterministic multithreading through schedule relaxation. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, pages 337–351, Oct. 2011.

[31] H. Cui, J. Wu, C.-C. Tsai, and J. Yang. Stable deterministic multithreading through schedule memoization. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, Oct. 2010.

[32] D. Dean and A. Stubblefield. Using client puzzles to protect tls. In *Proceedings of the 10th conference on USENIX Security Symposium - Volume 10*, SSYM'01, pages 1–1, Berkeley, CA, USA, 2001. USENIX Association.

[33] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: deterministic shared memory multiprocessing. In *Fourteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '09)*, pages 85–96, Mar. 2009.

[34] C. Dixon, T. Anderson, and A. Krishnamurthy. Phalanx: withstanding multimillion-node botnets. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 45–58, Berkeley, CA, USA, 2008. USENIX Association.

[35] EncFS. www.arg0.net/encfs, 2010.

[36] W. Enck, P. Gilbert, B. gon Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Symposium on Operating Systems Design and Implementation*, 2010.

[37] P. Ferguson and D. Senie. Network ingress filtering: Defeating denial of service attacks which employ ip source address spoofing. RFC 2827 (Best Current Practice), May 2000. Updated by RFC 3704.

[38] I. Fette and A. Melnikov. The WebSocket Protocol. RFC 6455 (Informational), Dec. 2011.

[39] E. Gessiou, V. Pappas, E. Athanasopoulos, A. D. Keromytis, and S. Ioannidis. Towards a universal data provenance framework using dynamic instrumentation. In *Information Security and Privacy Research - 27th IFIP TC 11 Information Security and Privacy Conference, SEC 2012, Heraklion, Crete, Greece, June 4-6, 2012. Proceedings*, pages 103–114, 2012.

[40] M. Gilliard. Live migration at the hp public cloud.

[41] Z. Guo, C. Hong, M. Yang, D. Zhou, L. Zhou, and L. Zhuang. Rex: Replication at the speed of multi-core. In *Proceedings of the 2014 ACM European Conference on Computer Systems (EUROSYS '14)*, page 11. ACM, 2014.

[42] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: Cold boot attacks on encryption keys. In *Proc. of USENIX Security*, 2008.

[43] M. R. Hines, U. Deshpande, and K. Gopalan. Post-copy live migration of virtual machines. *SIGOPS Oper. Syst. Rev.*, 43(3), July 2009.

[44] M. R. Hines and K. Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, New York, NY, USA, 2009. ACM.

[45] N. Hunt, T. Bergan, , L. Ceze, and S. Gribble. DDOS: Taming nondeterminism in distributed systems. In *Eighteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '13)*, pages 499–508, 2013.

[46] Imperva. Consumer password practices. www.imperva.com/docs/WP_Consumer_ Password_Worst_Practices.pdf, 2010.

[47] Intel Corporation. Laptop security with Intel Anti-Theft technology. www.intel.com/ content/www/us/en/architecture-and-technology/anti-theft/anti-theft- general-technology.html, 2012.

[48] Q. Jia, H. Wang, D. Fleck, F. Li, A. Stavrou, and W. Powell. Catch me if you can: A cloud-enabled DDoS defense. In *Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2014.

[49] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, M. Dahlin, et al. All about eve: Execute-verify replication for multi-core servers. In *Proceedings of the Tenth Symposium on Operating Systems Design and Implementation (OSDI '12)*, volume 12, pages 237–250, 2012.

[50] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, CCS '03, pages 272–280, New York, NY, USA, 2003. ACM.

[51] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis. Libdft: Practical dynamic data flow tracking for commodity systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, VEE '12, pages 121–132, New York, NY, USA, 2012. ACM.

[52] A. D. Keromytis, V. Misra, and D. Rubenstein. Sos: Secure overlay services. In *Proceedings of ACM SIGCOMM*, pages 61–72, 2002.

[53] G. Kontaxis, E. Athanasopoulos, G. Portokalidis, and A. D. Keromytis. Sauth: Protecting user accounts from password database leaks. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 187–198, New York, NY, USA, 2013. ACM.

[54] G. Kontaxis, M. Polychronakis, and A. D. Keromytis. Computational decoys for cloud security. In *Proceedings of the ARO Workshop on Cloud Security*, pages 261–270. Springer, 2013.

[55] A. Ku. Amazon Silk: Assisted Web Browsing (Sort Of). tom's hardware http://www.tomshardware.com/reviews/amazon-kindle-fire- review,3076-7.html, November 2011.

[56] G. T. Lakshmanan, F. Curbera, J. Freire, and A. P. Sheth. Guest editors' introduction: Provenance in web applications. *IEEE Internet Computing*, 15(1):17–21, 2011.

[57] L. Lamport. Paxos made simple. http://research.microsoft.com/en-us/um/people/lamport/pubs/paxos-simple.pdf.

[58] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. ACM*, 21(7):558–565, 1978.

[59] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[60] libevent. libevent.org/, 2015.

[61] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *International Conference on Architecture Support for Programming Languages and Operating Systems*, 2000.

[62] X. Liu, X. Yang, and Y. Lu. To filter or to authorize: network-layer dos defense against multimillion-node botnets. In *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, pages 195–206, New York, NY, USA, 2008. ACM.

[63] X. Liu, X. Yang, and Y. Xia. Netfence: preventing internet denial of service from inside out. In *Proceedings of the ACM SIGCOMM 2010 conference on SIGCOMM*, SIGCOMM '10, pages 255–266, New York, NY, USA, 2010. ACM.

[64] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[65] R. Mahajan, S. M. Bellovin, S. Floyd, J. Ioannidis, V. Paxson, and S. Shenker. Controlling high bandwidth aggregates in the network. *ACM Computer Communication Review*, 32:62–73, 2002.

[66] A. Mahimkar, J. Dange, V. Shmatikov, H. Vin, and Y. Zhang. dfence: Transparent network-based denial of service mitigation. In *NSDI*, 2007.

[67] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: building efficient replicated state machines for wans. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, volume 8, pages 369–384, 2008.

[68] D. Mazieres. Paxos made practical. Technical report, Technical report, 2007. http://www.scs. stanford. edu/dm/home/papers, 2007.

[69] T. Micro. Russian underground 101. http://www.trendmicro.com/cloud-content/us/pdfs/security-intelligence/white-papers/wp- russian-underground-101.pdf, 2012.

[70] Microsoft Corporation. Windows 7 BitLocker executive overview. technet.microsoft.com/en-us/library/dd548341(WS.10).aspx, 2009.

[71] G. Miło´s, D. G. Murray, S. Hand, and M. A. Fetterman. Satori: Enlightened page sharing. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, USENIX'09, Berkeley, CA, USA, 2009. USENIX Association.

[72] K.-K. Muniswamy-Reddy, P. Macko, and M. Seltzer. Provenance for the cloud. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST'10, pages 15–14, Berkeley, CA, USA, 2010. USENIX Association.

[73] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. In *Fourteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '09)*, pages 97–108, Mar. 2009.

[74] V. Pappas, V. P. Kemerlis, A. Zavou, M. Polychronakis, and A. D. Keromytis. Cloudfence: Data flow tracking as a cloud service. In *Research in Attacks, Intrusions, and Defenses - 16th International Symposium, RAID 2013, Rodney Bay, St. Lucia, October 23-25, 2013. Proceedings*, pages 411–431, 2013.

[75] B. Parno, D. Wendlandt, E. Shi, A. Perrig, B. Maggs, and Y.-C. Hu. Portcullis: Protecting connection setup from denial-of-capability attacks. In *Proceedings of the ACM SIGCOMM*, August 2007.

[76] P. A. H. Peterson. Cryptkeeper: Improving security with encrypted RAM. In *Proc. of the IEEE International Conference on Technologies for Homeland Security (HST)*, 2010.

[77] Ponemon Institute. The lost smartphone problem. www.mcafee.com/us/resources/reports/rp-ponemon-lost-smartphone-problem.pdf, 2011.

[78] G. Portokalidis and A. D. Keromytis. REASSURE: A self-contained mechanism for healing software using rescue points. In *Proceedings of the 6th International Conference on Advances in Information and Computer Security*, IWSEC'11, pages 16–32, Berlin, Heidelberg, 2011. Springer-Verlag.

[79] N. Provos. Encrypting virtual memory. In *Proc. of USENIX Security*, 2000.

[80] J. Rao, E. J. Shekita, and S. Tata. Using paxos to build a scalable, consistent, and highly available datastore. *Proc. VLDB Endow.*, Jan. 2011.

[81] J. Robertson. Security chip that does encryption in PCs hacked. www.usatoday.com/tech/news/computersecurity/2010-02-08-security-chip-pc-hacked_N.htm, 2010.

[82] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the migration of virtual computers. In *Proceedings of the 5th Symposium on Operating Systems Design and implementationCopyright Restrictions Prevent ACM from Being Able to Make the PDFs for This Conference Available for Downloading*, OSDI '02, New York, NY, USA, 2002. ACM.

[83] M. Savage. NHS 'loses' thousands of medical records. www.independent.co.uk/news/uk/politics/nhs-loses-thousands-of-medical-records-1690398.html, 2009.

[84] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. D. Keromytis. ASSURE: Automatic software self-healing using rescue points. *SIGPLAN Not.*, 44(3):37–48, Mar. 2009.

[85] A. Stavrou and A. D. Keromytis. Countering dos attacks with stateless multipath overlays. In *Proceedings of the 12th ACM conference on Computer and communications security*, CCS '05, pages 249–259, New York, NY, USA, 2005. ACM.

[86] R. Stone. Centertrack: an ip overlay network for tracking dos floods. In *SSYM'00: Proceedings of the 9th conference on USENIX Security Symposium*, pages 15–15, Berkeley, CA, USA, 2000. USENIX Association.

[87] Symantec Corporation. PGP whole disk encryption. www.symantec.com/whole-disk-encryption, 2012.

[88] Y. Tang, P. Ames, S. Bhamidipati, A. Bijlani, R. Geambasu, and N. Sarda. CleanOS: Mobile OS abstractions for managing sensitive data. In *Symposium on Operating Systems Design and Implementation*, 2012.

[89] A. Technologies. Q2 2015 state of the internet report, 2015.

[90] R. Van Renesse and D. Altinbuken. Paxos made moderately complex. *ACM Computing Surveys (CSUR)*, 47(3):42:1–42:36, 2015.

[91] I. VMware. Vmware distributed resource scheduler (drs).

[92] http://software.intel.com/en-us/intel-vtune-amplifier-xe/.

[93] B. Waters, A. Juels, J. A. Halderman, and E. W. Felten. New client puzzle outsourcing techniques for dos resistance. In *Proceedings of the 11th ACM conference on Computer and communications security*, CCS '04, pages 246–256, New York, NY, USA, 2004. ACM.

[94] S. Whalen, N. Boggs, and S. J. Stolfo. Model aggregation for distributed content anomaly detection. In *Workshop on Artificial Intelligence and Security (AISec)*, 2014.

[95] A. Whitaker, R. S. Cox, M. Shaw, and S. D. Gribble. Constructing services with interposable virtual hardware. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1*, NSDI'04, Berkeley, CA, USA, 2004. USENIX Association.

[96] J. Wu, Y. Tang, G. Hu, H. Cui, and J. Yang. Sound and precise analysis of parallel programs through schedule specialization. In *Proceedings of the ACM SIGPLAN 2012 Conference on Programming Language Design and Implementation (PLDI '12)*, pages 205–216, June 2012.

[97] A. Yaar, A. Perrig, and D. Song. Siff: A stateless internet flow filter to mitigate ddos flooding attacks. In *IEEE Symposium on Security and Privacy*, pages 130–143, 2004.

[98] J. Yang, H. Cui, J. Wu, Y. Tang, and G. Hu. Determinism is not enough: Making parallel programs reliable with stable multithreading. *Communications of the ACM*, 2014.

[99] X. Yang, D. Wetherall, and T. Anderson. Tva: a dos-limiting network architecture. *IEEE/ACM Trans. Netw.*, 16(6):1267–1280, 2008.

[100] A. Zavou, V. Pappas, V. P. Kemerlis, M. Polychronakis, G. Portokalidis, and A. D. Keromytis. Cloudopsy: An autopsy of data flows in the cloud. In *Human Aspects of Information Security, Privacy, and Trust - First International Conference, HAS 2013, Held as Part of HCI International 2013, Las Vegas, NV, USA, July 21-26, 2013. Proceedings*, pages 366–375, 2013.

[101] H. Zhao, A. Tang, N. Boggs, and S. J. Stolfo. Towards the detection of multistage attacks using cross-layer alerts correlation. Under submission.

LIST OF KEY ABBREVIATIONS AND ACRONYMS

ARC	application request cache
AWS	Amazon Web Services
CIFT	cloud information flow tracking
CRP	cascading rescue point
DDoS	distributed denial of service
DFT	data flow tracking
DMCC	distributed monitoring and cross-check
DMT	deterministic multi-threading
EC2	elastic compute cloud
EiGC	evict-idle garbage collector
EPC	enhanced post copy
GC	garbage collector
IFT	information flow tracking
MEERKATS	maintaining enterprise resiliency via kaleidoscopic adaptation and transformation of Software Services
MiSS	migration of software services
MOTAG	moving target defense mechanism against Internet DDoS attack
MRC	mission-oriented resilient cloud
OS	operating system
QoS	quality of service
RP	rescue point
SAuth	synergistic authentication
SDO	sensitive data object
SMR	state machine replication
StableMT	stable multi-threading
VM	virtual machine
VMM	virtual machine monitor